AFIT/GCS/ENG/93D-25

DOMAIN ANALYSIS AND MODELING OF A MODEL-BASED SOFTWARE EXECUTIVE

THESIS

Robert Lawrence Welgan
Captain, USAF

AFIT/GCS/ENG/93D-25

Approved for public release; distribution unlimited

AFIT/GCS/ENG/93D-25

DOMAIN ANALYSIS AND MODELING OF A MODEL-BASED SOFTWARE EXECUTIVE

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Robert Lawrence Welgan, B.S.C.S.

Captain, USAF

December, 1993

Acknowledgements

There are several people who supported me during this research. First, I would like to thank my thesis advisor, Dr. Hartrum for his sage advice. I also extend thanks to the other members of my thesis committee: Major Bailor and Major Luginbuhl. Their insightful and helpful comments during the review of this document improved its quality greatly. All three of them conspired to improve my briefing ability, too. Most importantly, I would like to thank my wife Stephanie. Her love and support gave me the strength to do my best.

Robert Lawrence Welgan

Table of Contents

List of Figures

List of Tables

AFIT/GCS/ENG/93D-25

Abstract

This research adapted the domain analysis techniques of Prieto-Díaz and Tracz to specify a domain analysis process which was used to conduct domain analysis over the domain of software executives. This analysis created a set of informal and formal domain model artifacts. The domain model artifacts were instantiated into two application executive subsystems. These executive subsystems operated in Architect, a domain-oriented application composition system based on the Object-Connection-Update (OCU) model. This research demonstrated and evaluated execution of the instantiated executive domain model in a series of event-driven and time-driven applications. As a consequence of developing the application executive for Architect, this research proposes additions to the OCU model.

DOMAIN ANALYSIS AND MODELING OF A MODEL-BASED SOFTWARE EXECUTIVE

1. Introduction

1.1 Background

In the past, computer scientists developed software based upon experience and the requirements of the system which was to use the software. Each time they began a project, they approached it anew. They were unable to capitalize on successful past development efforts unless they were members of those successful development teams. As software systems increased in size and complexity, this method of software development begot software riddled with errors. Some of these errors resulted in potentially life-threatening situations (22:2). Researchers at the Air Force Institute of Technology (AFIT) think that introducing the same kind of formalism that hardware engineers use into the software development process will lead to more reliable computer systems.

Today, AFIT computer scientists are experimenting with a whole new way of developing software: a model-based approach. This technique encourages software engineers to view software development in the same way that hardware engineers view hardware development. Hardware engineers re-use models of systems to construct physical devices. They employ practices which are common knowledge among engineers to trade off model parameters and arrive at a design solution before building the hardware. The application of hardware engineering principles of design reuse to software development is a central theme of software engineering. Model-based software development will help people write correct, reliable programs.

Hardware engineers rely on knowledge gained in previous development efforts. For example, each time an engineer builds a bridge, toaster, or car, he or she consults a design book containing generic models with parameters. Unless the engineer is building a product

based on radically new technology, other engineers have encoded the results of previous basic research and past implementations of these products in these models. Each engineer does not go into the lab to perform basic research on bridge, toaster, or car technology. The engineer conducts analysis based on the project requirements to determine the appropriate model parameter values. The engineer draws the design on paper or encodes it in a simulation using the parameterized models. The engineer's company tests the design before it commits resources to build the product. This method is seldom followed during software development. A software company builds its products without relying upon codified results of previous design efforts.

The Software Engineering Institute studied the idea of model based software development in their Software Architectures Engineering (SAE) project. They developed the object-connection-update (OCU) model which describes a set of reusable software building blocks (12). These building blocks can combine to form system models — thus storing knowledge gained about successful implementations of programs.

The OCU model describes systems which are comprised of subsystems. According to the SAE project team, the OCU model consists of the following elements that are repeatedly applied to patterns of requirements to express a design:

- controller - an entity which changes the state of objects
- import area - a location for input data
- export area - a location for export data
- object - an abstract representation of a real-world entity
- I/O device handler - a means to communicate with host hardware
- (monitor and control) surrogate - an interface between a controller and the host hardware
- executive - a program which supervises application execution (12:17)

An application developer combines the controller, object, import area, and export area elements to form subsystems. When the executive triggers the controller, the controller senses data placed in its import area, reacts based on that data and its current state, and places data in its export area. When each subsystem reacts, it activates one of possibly many update functions which changes the state of one or many subordinate objects

or subsystems. The executive determines which of the many possible update functions is actually used. The executive also determines the mapping between import areas, export areas, and I/O devices. Although SAE project members have not defined an executive, it plays a key role in OCU model operation.

The Joint Modeling and Simulation System (J-MASS) Program an Air Force response to the work done by the SAE Project. J-MASS project members intend to apply a version of the OCU design methodology to create and assemble simulation applications. The Air Force will eventually use these models to construct all Air Force simulations (27:6). Software developers will be able to create and modify multiple simulations simply by altering the interaction between predefined modeling components.

The Knowledge-Based Software Engineering (KBSE) Group at the Air Force Institute of Technology has developed an application composition and generation system called Architect. Architect is similar to the J-MASS system in that both of their structures are based on the OCU model. Architect differs from the J-MASS system in that Architect is designed to allow users to create model components from domain artifacts modeled as OCU primitives and combine them to form executable models. On the other hand, J-MASS requires modelers to use a J-MASS-specific set of modeling constructs (players, etc.) to describe an application component. Both of these systems allow application developers to choose from a group of pre-defined model components from one or many areas of interest (domains). Then, they can view the prospective model's behavior while the design is still "on-paper." If the model behaves correctly, an application developer may store this model back into the database of model components. (See Figure 1 (26).)

1.2 Problem

Previous research focused on the ability of the Architect system's underlying software architecture to generate domain-independent software specifications (2, 18). The underlying architecture, the OCU model as defined by the Software Engineering Institute, is incomplete. Prior to this research it did not contain a fully defined executive module. Indeed, OCU developers had not yet specified the way each subsystems' update function could acquire input, execute, generate output, or contend for host machine oper-

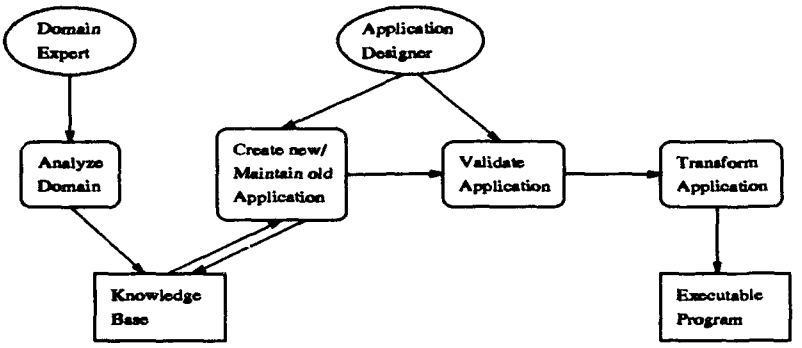

Figure 1. Automatic Software Generation

ating system resources during model execution (12). Furthermore, the previous Architect implementation of the OCU model also lacked all but the most rudimentary application executive. Prior to this effort Architect lacked:

- Executive control of entry of data into subsystem import/export areas
- Ability to account for delay during update function execution either in simulation time or in real time
- Ability to identify some subsystem input data as external to the simulation
- A concept of time — either real time or simulation time (18:6–10)

Although the KBSE group had partially tested Architect, full Architect system testing was not complete (18:6-6). The domains used to test the Architect system were not completely representative of all the types of simulations that are possible to execute with the system (18:6-6). The domains implemented by Anderson and Randour contained objects that could only execute in a non-event-driven sequential mode of execution. In other words, simulation entities were updated in a fixed order during each execution. Architect did not permit an application specialist to define event-driven or time driven simulations. For a system like Architect to be useful to the Air Force, it must be able to run many types of applications. Architect needed a more powerful executive to model these application behavior in these execution modes.

Problem Statement

The purpose of this research was to specify, implement, and analyze an application executive subsystem model which allocates host machine and subsystem resources during application execution.

1.3 Scope

This research defined an application executive for the Architect implementation of the OCU model only. It did not involve converting Architect's structure to conform to any other model-based software development architecture. This research lead to extensions to the OCU model to make the implementation of an application executive possible. The wide spectrum language REFINE[1], together with the object and dynamic models popularized by Rumbaugh (11), were used to specify an application executive domain model. This research incorporated time as an attribute of the overall composed application. The application executive complemented the KBSE group's domain models, which were developed concurrently with the application executive. During this research, the addition of an application executive to the Architect system resulted in changes and additions to the Architect domain model definition. The application executive built here does not meet real-time system constraints. The application executive developed here does not execute simulations where application subsystems are created and destroyed at run time, because its proper operation does not depend on the existence of a fixed number of subsystems during run-time. However, this research did not address the necessary changes to the Architect implementation of the OCU model that would allow an application specialist to create applications which contain components that are created and destroyed dynamically. Finally, the application executive model does not allocate application subsystems to multiple processors.

1.4 Approach

The definition of an application executive subsystem for any composed software model in Architect required completion of the following steps:

[1] REFINE^{TM} is a trademark of Reasoning Systems, Inc.

1. *Define Domain Analysis Process* - In order to create a domain model, domain analysis must be performed. The first step in this research was to define a domain analysis process that created a domain model for the application executive.
2. *Perform Domain Analysis* - After the domain analysis process was defined, the next step in this effort was to carry out this process and produce a domain model.
3. *Instantiate Domain Model* - The third phase of this research was the creation of an instance of the application executive domain model. Tests recorded the behavior of each application executive instance.
4. *Analyze Executive* - The final phase of this research involved studying the operation of the application executive and analyzing the instantiated domain model test results together with the process used to create the domain model.

1.5 Assumptions

During the specification and implementation of an application executive for Architect, it was assumed that each application subsystem would be passive and not call its update function unless told to do so by the executive. The research assumed the application executive would not handle object management, but constructs from the domain model implementation language (REFINE) would manage them instead. A goal of this research was to develop an executive that could run applications constructed using new domain models simultaneously developed by other researchers as well as applications in the old domains defined for the previous version of Architect.

1.6 Sequence of Presentation

Chapter 2 contains the results of a literature review over topics relating to supervisory programs, the OCU model, and the specification and implementation of concurrent programs. Chapter 3 defines the executive domain analysis process and a summarizes the informal application executive domain model. Chapter 4 describes the details of the transformation of the application executive domain model into REFINE constructs and the way the model was incorporated into the OCU paradigm. A discussion of the way the domain

model primitives were instantiated and included in a composed application appears in Chapter 5. Chapter 6 discusses the executive test procedure and its results. Conclusions and recommendations for further model enhancement appear in Chapter 7. Appendix A contains the detailed results of informal and formal executive domain analysis. Appendix B shows the instantiated domain model test results.

2. *Survey of Current Literature*

2.1 Literature Review Goals

This review focuses on the characteristics of supervisory and executive programs which are similar to Architect's application executive. The application executive, like any other supervisory program, is a resource manager for the rest of its associated application. This review covers applicable features of resource management software. As a simulation executive, the Architect application executive must be able to model time in applications that simulate time-passage. While this research is not concerned with defining a real-time application executive for Architect, some concepts associated with specifying real-time systems may carry over to the specification of a non-real-time application executive. For example, some subsystems may execute concurrently. Architect's structure is based on the Object Connection Update (OCU) model (12), and the OCU model expects certain things from an application executive. These expectations figure prominently in the design and implementation of the executive. This review discusses related research. The U.S. Air Force Joint Modeling and Simulation System (J-MASS) contains an Execute Simulation subsystem. A description of the services supplied by this subsystem lends some insight into the services required by an Architect application executive. One of the products of this research is a special model of an application executive called a domain model. Domain modeling and analysis is a good place to begin the review.

2.2 Domain Modeling

A domain is simply an area of study. For example, if one were an expert saxophone maker, he would be a domain expert in saxophones. Applicable concepts in the domain of saxophones might include springs, pads, reeds, and brass. Springs constitute a separate domain of their own. A domain model of a saxophone may include all of these things, or it may include only one of these things broken down into its constituent parts. It is up to the domain engineer to determine the level of abstraction in the domain model. Supervisory programs and computer operating system kernels are this research's applicable domains.

A domain model is the result of domain analysis. Lowry describes domain analysis as "...a form of knowledge acquisition in which concepts and goals of an application domain are analyzed and then formalized in an application oriented language suitable for expressing software specifications."(14:648) Lowry does not provide his readers with a method for performing domain analysis, but Prieto-Díaz does (17). Prieto-Díaz is primarily concerned with the same area of interest as the builders of Architect—software reuse.

Prieto-Díaz in (16) studied various domain analysis techniques. The results of his study of domain analysis by Raytheon, the Common Ada Missile Project (CAMP), McCain, and Arango are summarized in Figure 2.

Raytheon	CAMP	McCain	Arango
Identify commom functions Group by classes Organize into library Analyze business system structures Identify common structures Abstract structures Build objects	Identify similar systems Decompose functionally Abstract functionally Define interfaces Encapsulate results	Define reusable entities Define resuable abstractions Classify abstractions Encapsulate results	Bound domain Collect examples Identify abstractions Formalize abstractions Classify abstractions Encapsulate results

Figure 2. The Domain Analysis Methods Compared by Prieto-Díaz

Prieto-Díaz used common elements from these domain analysis techniques to form his own domain analysis process. The steps in his domain analysis technique are:

1. Pre-Domain Analysis
   (a) Define the Domain
   (b) Scope the Domain
   (c) Identify Sources of Domain Knowledge
   (d) Define Domain Analysis Goals and Guidelines
2. Domain Analysis
   (a) Identify Objects and Operations
   (b) Abstract the Objects and Operations
   (c) Classify the Abstracted Objects and Operations
3. Post-Domain Analysis

(a) Encapsulate the Classified Objects and Operations

(b) Produce Reusability Guidelines (16:66)

Prieto-Díaz's domain analysis technique is analogous to the development of abstract data types. An abstract data type is a set of entities of interest (objects) and operations upon those objects (methods). Similarly, domain analysis requires one to reason about an area of interest (a domain) and to group or encapsulate operations on those entities of interest into a structure. This structure becomes a domain model. In order to group entities into a domain, Prieto-Díaz says that one must determine the boundaries of the domain (16:64). Using these boundaries, one must apply a classification scheme to the entities in these boundaries. The result is a group of objects related by common operations that can be performed upon them. Also, Prieto-Díaz's method requires a domain analyst to express his domain as a group of related objects in the same way as Rumbaugh elicits an object-oriented design from a problem specification (11). The similarity between Prieto-Díaz's domain analysis technique and the development of Rumbaugh object models suggests that perhaps Rumbaugh object models are an appropriate way to express a domain model.

Tracz, Conglianese, and Young build upon the work of Prieto-Díaz to define a specific domain analysis technique. They state that there are two sides to the domain analysis process: analysis of the problem space, and analysis of the solution space (28:2). Prieto-Díaz is primarily concerned with analysis of the solution space. His domain analysis process relies on the study of previously implemented solutions. Tracz is concerned with a study of the problem space. He mentions that domain analysis must be concerned with system constraints imposed by the real world (28:3). Most all domains must consider real-world constraints, especially those constraints which relate the domain model's dynamic behavior. Including these constraints is an important step in the any domain analysis process.

The DIALECT[1] User's Guide (19:4-1–4-12) discusses another domain analysis process with an emphasis on the resulting domain model structure. The DIALECT approach requires the domain analyst to configure the entities of interest into an abstract syntax tree structure based upon the relationships between the entities. In a Dialect domain model,

[1] DIALECT is a registered trademark of Reasoning Systems, Inc.

the entities are called objects, and the relationships are called attributes. This tree structure permits the analyst to traverse the tree and perform operations on the objects at each node of the tree. These operations can transform the tree into executable software.

2.3 Resource Management

Architect's application executive must oversee the operation of all software routines in its associated model. In this way, the application executive is functionally similar to a computer operating system kernel. Simply stated, an operating system provides an interface between computer programs and the computer hardware itself. Operating systems define abstractions of the computer resources including the processor, memory, and external devices. Computer programmers interact with these abstractions when they write programs. An understanding of basic computer operating system functions leads to an understanding of some necessary characteristics of an application executive.

Operating systems have existed in some form since the 1950's, and there are numerous books and articles which describe their mechanisms in detail (1, 9). The characteristics that are relevant for this research involve controlling and specifying sequential and concurrent execution of software routines. Other important characteristics include sharing internal resources (data) and external resources (files, printers, and the like) during program execution. Andrews and Schneider divided languages which specify concurrent execution into three classes: procedure oriented, message oriented, and operation oriented (9:37). Procedure oriented languages rely on procedure calls to operate on objects. Message oriented languages assign a process called a caretaker to each object. Messages to these caretakers are the only way to access object state information. Operation oriented schemes also rely on message passing. However, operation oriented languages combine aspects of procedure oriented and message oriented languages in that each object has a caretaker associated with it but it is accessed by a procedure call (9:37).

2.4 Concurrency and Temporal Programming

This research involves specifying an application executive for Architect which serializes execution events. Levi and Agrawala define an event as a "detectable, instantaneous,

and atomic change in system state." (13:15) Richard Fugimoto summarized the different methods of implementing an event-driven simulator in (7). Alagar and Ramanathan describe axioms and theorems used to prove correct operation of software which controls reactive systems (23). Reactive systems are those real-time systems that operate by reacting to internal and external events. Alagar and Ramanathan state and prove nine theorems describing the relationship between tasks and events in an automated system. These theorems are used to prove the properties of an abstract algebra on the set of events E. Finally, they specify system constraints on a robot assembler of cups and saucers in terms of this algebra, and prove the correctness of their specification.

Alagar and Ramanathan describe events in a reactive system by zeroing in on an essential quality of events: duration. They show the concept of duration in an easily understood way. They depict each event as a directed line. Overlapping lines in the diagram show their overlap in time (Figure 3).

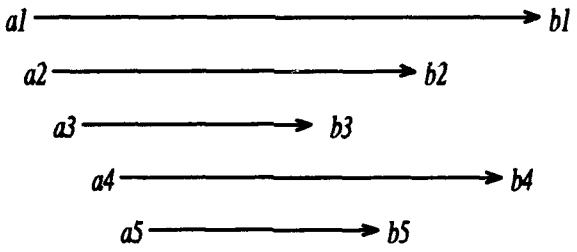

Figure 3. Events in E

Mathematically, these relative start and stop times are shown as sequences. Figure 3 describes the following two sequences of start and stop times:

$$TIME_1(e) = \{a_1, a_2, a_3, a_4, a_5\}$$
$$TIME_2(e) = \{b_3, b_5, b_2, b_4, b_1\}$$

$TIME_1$ lists relative start times for each event and $TIME_2$ denotes relative stop times for each event. Thus, in Figure 3 event a_1 started first, but the third event a_3 finished first as shown by point b_3 in the diagram.

The method of specifying events as a duration, as proposed by Alagar and Ramanathan, may provide a means for an application executive to ensure that no causality errors occur during system operation. A causality error occurs when a subsystem executes based on information from another subsystem which may have executed too early, possibly corrupting this information. If the application executive knows how long an application subsystem will execute, it can encode this information in the event as it is scheduled, or the subsystems that raise events can include the event duration as they raise them. When events include a duration, the executive can predict when this application subsystem will stop raising events. If the executive knows when all subsystems will stop raising events at a particular time t, it can determine when it is safe to increment the clock. If an executive specifies events as points in time and not as a duration in time, then, as long as the subsystem notifies the executive when it is finished using the processor, the executive can calculate which subsystems need to use the processor at a particular time. Specifying events as points in time and durations of time both allow the executive to decide which model subsystems can execute concurrently. Fugimoto calls the technique of only allowing parallel subsystems to execute when there is no chance of a causality error occurring a conservative approach (7:6). The conservative technique is not without disadvantages. For instance, a conservative approach relies on the application programmer's ability to predict which events should precede other events before the software executes (7:16). If the application uses durational events, the application subsystem must be able to predict how long it will use the processor before it actually uses it. Similarly, if an application models events as points in time, it must predict when to schedule an event that signifies when each subsystem will finish using the processor before the subsystem begins using the processor.

In response to the criticisms of the conservative approach, researchers have devised an optimistic technique. Instead of preventing causality errors, systems using the optimistic technique sense them and recover(7:17). Fugimoto describes what he calls the most popular optimistic mechanism—Time Warp. Simply stated, Time Warp senses a causality error when an event which has a timestamp smaller than the processor timestamp executes. If this happens, the processor's state is restored to its state immediately preceding the timestamp of the process which caused the roll-back. This type of optimistic mechanism

presupposes that the processor keeps a record of the way its state changed over time up until the system's global time value. This time, called Global Virtual Time (GVT), is the value of the "smallest timestamped, unprocessed event."(7:17) The processor can discard all processor's states before GVT, because the processor will never roll back before GVT.

Optimistic mechanisms are not without their disadvantages as well. For example, they can become very complicated depending on the number of concurrent processes executing in the simulation. This complexity also makes them very data intensive, depending on the number of different states that must be kept for each process. The number of causality errors that occur during a simulation may cause the processes to roll back so many times that overall system performance would drop considerably. If there was a way to limit causality errors through a formal specification of event order, then GVT would be kept relatively large (recent) during the simulation. This would reduce both the data intensive and complex nature of the simulation. Fugimoto cites the efforts of others who have tried in a similar way to combine conservative and optimistic methods (7:21). Their efforts rely on the relationship between events which results in a varying amount of causality errors throughout program execution. This important choice of whether Architect should follow a conservative or optimistic approach occurs during domain analysis of the domain of application executives. The choice of how to specify events in Architect is made during domain analysis as well.

2.5 The Object-Connection-Update Model

The Carnegie-Mellon Software Engineering Institute's Object-Connection-Update (OCU) model provides an excellent way to model systems that are normally composed of subsystems. These subsystems are made up of controllers, import objects, export objects, and primitive objects. Controllers signal their child objects to update when they are themselves signalled by the application executive. (See Figure 4 (12).)

An executive, also modeled as a subsystem, signals all the subsystems in the simulation for the purpose of controlling application execution. D'Ipollito wrote about an example of one such system in (6:260). In this example, each subsystem in an aircraft engine is modeled as an OCU subsystem, and the executive encapsulates all the subsystems.

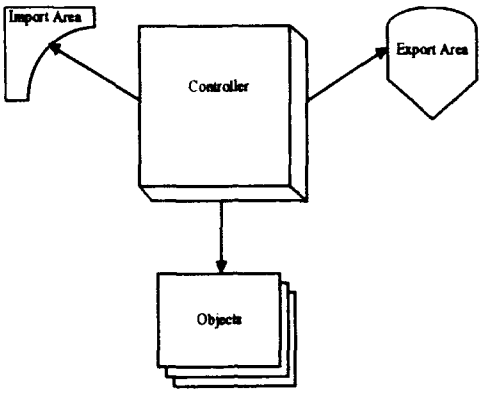

Figure 4. OCU Subsystem Model

In (12), Lee discusses the OCU model and its operation. He mentions the operation of the application executive only parenthetically. The section of the draft document describing the executive is not yet part of the document. However, it is possible to infer something about the OCU application executive from the description of the OCU model in (12). For example, Lee mentions that each subsystem in the OCU model is passive and it waits for the application executive to tell it to execute its function(12:18). This means that the executive must have knowledge of each subsystem in the model. Also, the executive must decide when each subsystem should fire. According to Lee, the executive may direct each subsystem to take any of the following actions:

- Update - change state based on import area state data, export new data to export area
- Stabilize - normalize object state to system state
- Initialize - create sub-objects if necessary and external interfaces
- Configure - set characteristics of controller and object
- Destroy - eliminate associated objects (12:19)

The OCU executive activates each subsystem using the subsystem procedural interface. The OCU executive also controls data flow through the application. In order to get and place external data in the import areas of its subsystems, an OCU application executive must be able to control I/O Device Handlers. These structures exist in OCU for the purpose of sending and receiving data from external devices (12:23). The application executive must be able to map I/O Device Handlers to subsystems in the simulation.

Then, each subsystem will have access to the host systems services it needs during the simulation.

In an draft working paper, Bailor and members of the Software Engineering Institute developed a tentative structure of an OCU Executive (3). This paper described subsystems needed to permit any OCU-based system to subscribe to application executive services to allow the OCU-based software to accept external events, schedule processes, and allocate processes to distributed processors as necessary. Specifically, Bailor calls for an application executive to be made up of:

1. *Schedule Manager Subsystem* Makes and stores subsystem and hardware scheduling decisions based on host hardware constraints
2. *Event Manager Subsystem* Records and orders events for the application executive
3. *Import/Export Manager Subsystem* Controls all application import and export areas. Ensures that these areas are properly updated over time
4. *Control Sequencer Subsystem* Executes internal and external interfaces
5. *Registrar Subsystem* Records initial application-dependent initialization data for executive use during system execution (3:5–6)

A few things are unclear in this rather sparse description of an OCU application executive. How do these subsystems interact and what controls them? It appears from the description of the subsystems that they can interact concurrently and are driven by the event manager's service of each executive event. How does this concept express events? Are they durational or point events? This description of an OCU executive does not go into that level of detail. Finally, how do these subsystems maintain time? Bailor's concept does not provide a separate subsystem to maintain a global clock, but it does not preclude access to the system clock or prevent an application specialist from including a clock as a model component.

2.6 Joint Modeling and Simulation System

The U.S. Air Force is currently studying model-based software systems under the auspices of the Joint Modeling and Simulation System (J-MASS) program. The J-MASS concept contains an Execute Simulation subsystem which addresses many of the same problems associated with building an application executive subsystem for Architect.

2.6.1 J-MASS System Overview. The J-MASS simulation system allows a user to build a simulation from a library of existing model components (MC). The user may form experiments from many simulations. He or she can configure the system to record whatever data needed from the experiment. Finally, the user enters the Execute Simulation Mode. Details about the use and operation of the J-MASS system are contained in (21). This research is concerned with the way J-MASS designers chose to structure the Execute Simulation Mode software.

2.6.2 J-MASS Execute Simulation Software. The Execute Simulation software is broken down into a number of components referred to in (21:45-48) as agents. These agents carry out the following functions:

- *Browser Agent* Allows the user to select simulations and experiments to run.
- *Execution Agent* Serves as the intermediary between the user and the simulation. Allows the user to control and monitor the progress of the simulation.
- *Experiment Agent* Subscribes to host system hardware and software resources prior to simulation execution. Creates the proper number of Simulation Run-Time Agents.
- *Simulation Run-Time Agent* Acts as the provider of services to the simulation MC's. Controls progress of time, intermodule communication, and host resource allocation of the MC's. It is closest to the concept of what an application executive should be. As an analog of the Architect application executive, it deserves further study in the next section.

2.6.3 J-MASS Simulation Run-Time Agent. Before going into how the J-MASS designers structured the SRA, it is useful to look at some top-level design tradeoffs. Fortunately, the authors of (21) use an entire appendix to discuss one important design consideration for the Simulation Run-Time Agent (SRA): the difference between the modeling of control by a simulated system and controlling the simulation itself. The designers chose to eliminate any mingling between simulation control and model components for the reasons listed in Table 1 (21:D-1).

After concluding that the SRA should not be intermingled with any real-world model's simulated control functions, the authors describe the remaining two key concepts which guided their decisions regarding SRA structure: activation and control flow (21:D-2). They define activation as "...A conceptual event that corresponds to the initiation of

Embedding Control	Extracting Control
In real-world entity	Not in real-world entity
Depends on structure	Does not depend on structure
Has knowledge of other things	Knowledge is restricted
Limited sub-component Control	Unlimited sub-component control
Modules tightly coupled with MC	Loose coupling of control to MC

Table 1. The Differences Between Embedding and Extracting Control of a Simulation from the System Being Modeled

a cycle in which an entity updates its state."(21:D-2) Control flow is "a conceptual message conduit that provides instructions ... to the entity which determine the manner in which the entity's state is updated."(21:D-2) Thus, the design of the SRA focuses on the specification of control flow to determine the order in which MCs execute.

SRA designers were forced to deal with implementation issues resulting from their specification of activation and control flow. Specifically, they needed to find some way to implement conditional MC execution. They defined a grammar which can be evaluated under predicate logic to describe execution conditions(21:D-14–D-16). They also needed to find a way to notify MCs that they have been activated. SRA designers, having chosen a message-passing architecture for the SRA, relied on the concept of a daemon. A daemon is a system program which serves as a process that acts as a receiving connection(1:67).

With all these considerations in mind, an SRA consists of:

- *Simulation Controller* - controls SRA
- *Scenario Manager* - creates simulation objects using simulation script
- *Synchronizer* - controls simulation time and enforces MC timing constraints
- *Spatial System Manager* - keeps and updates module location data
- *Simulation Run-Time Interconnect* - allows multiple SRA's to communicate
- *Locus of Control* - collection of MC's with common thread of control
- *Controller* - coordinates LOC execution
- *Activator* - serializes MC execution
- *Data Management Package* - holds all simulation/execution data
- *Journalizer* - performs data reduction as specified by the user

- *Intercomponent Interconnect* - allows the model and executive to communicate with a given LOC (21:E-6)

The SRA parts listed above are important in that they provide a way to understand the basic services that the SRA provides to the rest of J-MASS. The Architect application executive provides the same sort of services. The domain analysis over the domain of application executives (see Appendix A) uses this information to determine which services an Architect application executive should perform.

2.7 Conclusion

Although the creators of the OCU model have not specifically defined their concept of an executive, a review of OCU literature helped define what the OCU paradigm requires from an executive. An overview of current domain analysis practices and the J-MASS program pointed the way to the definition of a domain analysis process for this research effort. It also led to the definition of an application executive domain model.

3. *Informal Domain Analysis*

3.1 Introduction

This chapter defines the domain analysis process used to create an informal domain model of an Architect application executive. Creating an informal domain model involved determining what the application executive was required to do by analyzing the executive capability prior to this effort, proposing additions or changes to that capability, and analyzing the domain of executive software with these limitations and additions in mind. Additions to the former application executive capability are proposed in accordance with the research goals and scope outlined in Chapter 1. This chapter describes the specific domain artifacts used to create an informal, general model of an application executive for Architect. A summary of the results of domain analysis together with the object model of an Architect application executive are presented, while the details of the domain analysis itself (including the object, dynamic, and functional parts of the domain model) appear in Appendix A.

3.2 Informal Specification of an Application Executive

Domain analysis over the domain of supervisory programs resulted in the creation of a domain model of an application executive. The application executive was informally specified as a list of executive services. The application executive domain model consists of objects and operations that provide these services. This section details the domain analysis process used to create a domain model of an Architect application executive, describes the artifacts used to express the domain model, and presents a summary of the resulting domain model.

3.2.1 Domain Analysis Technique. Chapter 2 discussed the concepts of domain analysis and domain modeling without giving specifics on how these concepts might be applied to the domain of executive programs. The purpose of this section is to define the domain analysis process used in this research. A detailed explanation of the domain analysis appears in Appendix A. The resulting domain model is presented in Section 3.2.2.

A process of reasoning about common characteristics of supervisory programs led to a domain model of an application executive. Prieto-Díaz agrees that this is a valid way to proceed. "This is the very process of domain analysis: identifying and storing information for reusability."(17:47) Chapter 2, which contains information about characteristics of supervisory programs and the specification of those characteristics, served as the basis of the domain analysis process itself. The major topics in this review resulted from a top-level study of executive software. Chapter 2 expresses enough information for the construction of the domain model. However, merely expressing the information is not enough to transform it into a domain model. This information must be structured and placed within the context of its reuse environment. "In domain analysis, experience and knowledge accumulates until it reaches a threshold. This threshold can be defined as the point where an abstraction can be synthesized and made available for reuse." (17:47) Prieto-Díaz cites numerous ways others proposed to conduct domain analysis (17:48–50). All these approaches share the ideas of specifying a level of abstraction and generalizing components of a domain into their most essential elements.

Ideas from Prieto-Díaz and Tracz, et al. combine to form the Architect Application Executive Domain Analysis Process, depicted in Figure 5. The domain analysis process drawn in Figure 5 adds real world implementation concerns to the Prieto-Díaz domain analysis process. It forces the domain engineer to consider the goals of domain analysis when he chooses the domain modeling language. The following is a breakdown of each of the steps in the Architect domain analysis process.

Name Domain Before domain analysis can begin, the domain analyst must name the domain under study. The domain under study is a reflection of project requirements. For the purposes of this effort research requirements dictate the domain of interest.

Scope Domain There is often a blurred line between two related domains. It is not always clear where one domain ends, and the other begins. During this step, the domain analyst decides where the domain under study begins and ends. Research requirements help the domain analyst determine the constraints placed upon the domain as a result of its need to fit into a particular system. The domain analyst

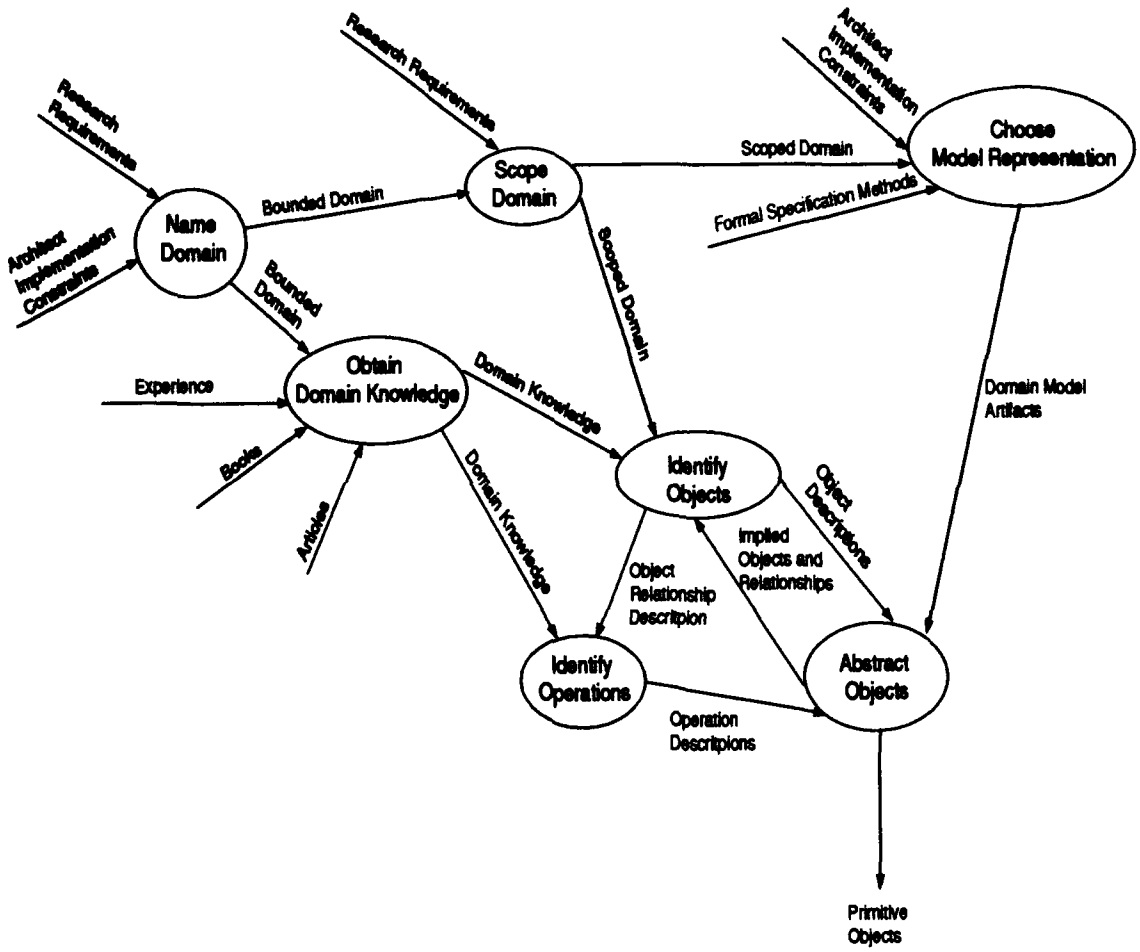

Figure 5. Architect Application Executive Domain Analysis Process

must balance the need to consider these constraints without paring down the domain merely for the sake of convenience.

Obtain Domain Knowledge The best person to perform domain analysis is an expert in that domain. If the domain analyst is not an expert in the domain he or she may need to consult persons knowledgeable in a particular domain to obtain enough information about the domain to reason about it in a general way. During this step, the domain analyst records key facts about the scoped domain and organizes his or her thoughts prior to listing domain objects and operations.

Choose Model Representation The result of the domain analysis process is a domain model. Prior to beginning domain modeling, it is necessary to decide on the best way to represent the desired domain model in order to provide the greatest ease of use. Since the domain model needed to be implemented in REFINE, REFINE was used as the formal specification language (See Appendix A for other details). However, selecting the formal specification language is only part of choosing the model representation. The other part involves describing the artifacts that will make up the informal domain model. The application executive domain model contains object-oriented analysis diagrams as specified in (11), as well as the formal portion. These diagrams allow the domain analyst to capture the structure of the objects that perform executive services as well as their interrelationships. These diagrams and Rumbaugh's object, attribute, and operation identification techniques help the domain analyst structure the resulting model. This eases the model's eventual transformation into executable REFINE code.

Identify Objects In this domain analysis process, the domain analyst views the entities of interest in the same way that Rumbaugh views objects. According to Rumbaugh "...an object...is a concept, abstraction, or thing with crisp boundaries and meaning for a problem at hand." (11:21) The domain analyst looks for entities in the domain of interest and describes them and their attributes with short prose statements as well as object diagrams as in (11).

Identify Operations Rumbaugh views operations in the context of how they change the state of objects. "An operation is a function or transformation that may be applied

to or by objects..." (11:25) Similarly, the domain analyst uses relationships between objects and his or her knowledge of how these objects behave in the domain to describe these functions or transformations. The domain analyst expresses the dynamic relationships between objects using the Rumbaugh dynamic modeling technique.

Abstract Objects While identifying objects in the domain and expressing them graphically, the domain analyst needs to express them more formally and to view them from varying levels of abstraction using the features of a formal specification language. The domain analyst augments the object and dynamic models with a functional description of objects in the domain. These functional descriptions are in terms of the precondition and postcondition for each object function. The domain analyst converts the textual and graphical descriptions into REFINE object and attribute declarations. During this transformation process, he or she may find errors in the object descriptions or may come up with new objects that need to be elaborated.

3.2.2 Domain Model. Appendix A contains the application executive domain model details. This section contains key domain analysis results that aid the reader in understanding the basic functionality of the Architect application executive and its primitive objects.

3.2.2.1 Application Executive Services. These are the services an application executive must provide to composed models, as determined by the results of the *Obtain Domain Knowledge* step of the application executive domain analysis process:

1. *Event Handling* - The executive services events raised by the application during execution. The executive orders events and may generate events for the application executive and model components.
2. *Registration* - The executive gets application-specific requests for executive services. This involves building executive data structures with information about application subsystems to enable the executive to keep track of each subsystem's execution state. For example, the executive subsystem collects the connections in the composed application and controls them throughout application execution.

3. *Activation* - The executive activates particular subsystems when it's time for them to execute. The order of activation is determined by the schedule and by the current execution state of the component that must execute. Activation is the result of an event and may cause other events to be generated.

4. *Communication* - The executive supervises internal application data transfer. It manages connections between components and provides inputs for processes when it is time for them to execute.

5. *I/O Handling* - The executive supervises application calls for external I/O services. It controls external device interaction with the application. It manages application access to device drivers and buffers for those devices.

6. *Scheduling* - The executive makes an execution schedule for the application. It orders events and determines how to serialize concurrent events.

3.2.3 Executive Mode Requirements. The application executive provides these services for four different types of applications. These types of applications differ in the way in which they execute. Different ways of executing are referred to as *modes*. The domain model considers these execution modes:

- Event-driven sequential
- Event-driven concurrent
- Time-driven sequential
- Time-driven concurrent

These mode descriptions are not necessarily mutually exclusive. Figure 6 displays the relationships between application execution modes in Architect. Applications employ a combination of modes which describe what causes a model subsystem to execute (drive), and how many model components can execute at one time (concurrency). During previous research, the KBSE group defined a simple application executive which operated in a non-event-driven sequential mode. Throughout the remainder of the document, when an example or a discussion refers to one mode characteristic (i.e. sequential mode) it

is referring to all possible modes in the same partition (in this case both time-driven sequential and event-driven sequential modes).

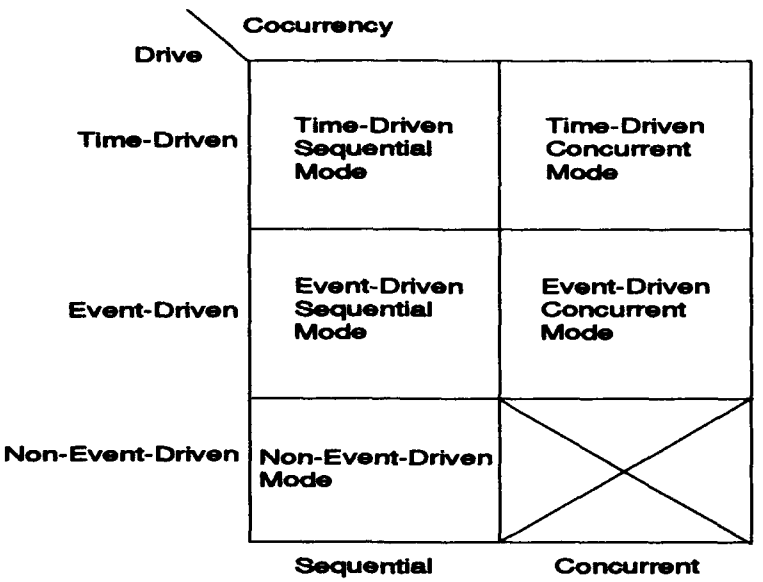

Figure 6. The Relationship Between Modes in Architect

An event-driven application executes as the result of executive service of events that are asynchronously raised by the application subsystems and the executive. A time-driven application contains subsystems that react to changes in the clock. A sequential application contains one thread of control. A concurrent application has multiple, simultaneous threads of control. The application executive provides its services in different modes by using different combinations of executive domain model primitives to handle these different modes of operation.

Figure 7 depicts the application executive domain model. These are the objects which provide the services listed in this section. Detailed definitions of these objects, attributes, and operations are in Appendix A

3.3 Domain Model Implementation Goals

The domain model described in Section 3.2.2 was transformed into instantiated REFINE primitive objects in the REFINE object base and executable functions also written

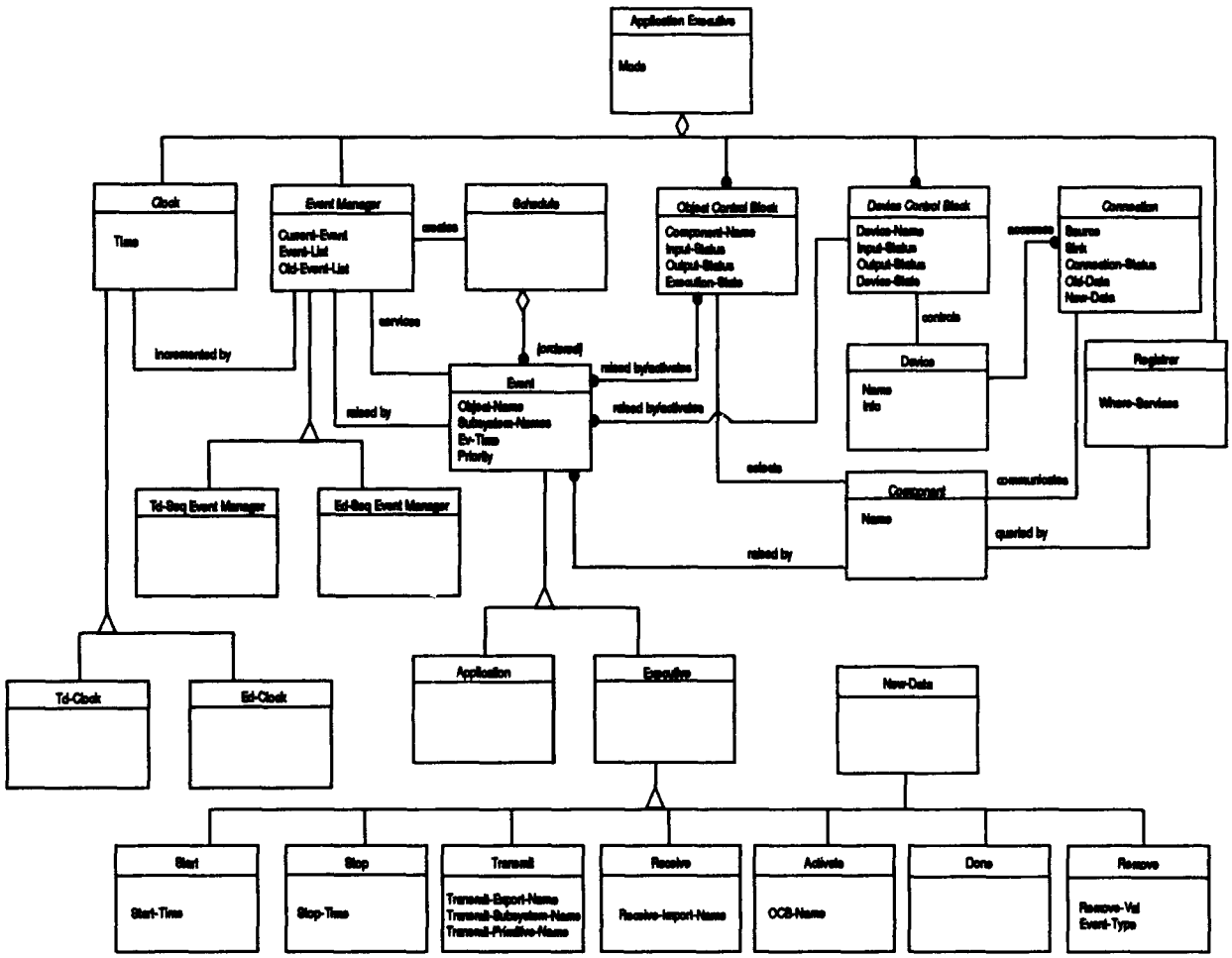

Figure 7. Application Executive Object Model

in REFINE. These REFINE functions manipulate the objects depicted in the object model and perform the executive services when requested by the application. Once an executive is instantiated it should not be reinstantiated each time a new application is built. Instead, the instantiation is expressed in a language that describes the domain of application executives, and the description of the instantiated application executive is saved in a file. The development of a domain specific application executive grammar is a necessary part of saving the Architect application executive so it can be brought into each new application as the application is composed.

The application executive can be structured in a number of ways, according to the OCU model. Chapters 1 and 2 discussed each OCU structure available. This research chooses from the following methods of structuring the domain model:

- A set of related subsystems that each perform an executive service.
- A single subsystem that acts as an executive.
- Some combination of the two.

Flexibility is another goal of the Architect application executive. After all, as the executive is used more and more, future users may want to augment it to change such things as its communication model or its method of scheduling events. The Architect application executive structure takes these considerations into account.

3.4 Conclusion

A study of previous application executive capabilities, current capabilities used for execution, and domain analysis led to an informal application executive domain model. This informal model was transformed into a formal model expressed as executable REFINE code and included in Architect. Chapter 4 describes how the informal domain model was transformed into the formal domain model. Chapter 5 shows how the primitives developed in Chapter 4 were combined to form instantiations of Architect's application executive subsystem.

4. Domain Model Formalization

4.1 Introduction

The previous chapter presented an informal domain analysis for the Architect application executive. It described the domain analysis method used to develop the domain model. Chapter 3 described how Architect's structure influenced the domain model representation technique. The application executive domain model consists of text, drawings, and REFINE object and function declarations. Although Chapter 3 made a effort at accounting for Architect's OCU basis, the informal domain model depicted there does not consist of OCU artifacts. This chapter describes the application executive domain model's formalization as a set of Architect-compliant primitives. It explains the executive's concept of operations during execution of non-event-driven, event-driven, time-driven, concurrent, and sequential applications. The suitability of REFINE as a formal modeling language is presented here as well.

4.2 The Role of REFINE in Formalization

REFINE is a wide-spectrum computer language. In other words, it can serve as a specification, design, and implementation language. It is an ideal way to express the informal domain model. It contains constructs called objects. An object is a data type in REFINE used to describe an entity of interest to the programmer. Objects have associated attributes that describe qualities of an object. REFINE allows the programmer to declare set theoretic data types and to reason about them using first order predicate calculus constructs. REFINE allows incremental development of specifications. Its predicate statements allow a software engineer to declare pre-conditions that must exist and post conditions that must be satisfied during an operation written in REFINE. This automatic predicate transform allows a developer to write REFINE code that emphasizes what the code needs to do instead of how the code should do it. These features of REFINE blur the line between analysis (the domain model) and design (the implementation). The application executive domain model was implemented using REFINE and relied on the predicate transforms to describe pre- and post-conditions discovered during domain analysis. A de-

tailed explanation of REFINE is beyond the scope of this chapter but can be found in (20) and (19).

4.3 Formalization Technique

The informal domain artifacts were transformed into formal domain artifacts known as primitives. These primitives correspond to the OCU concept of objects. These objects were constructed using the techniques described by Randour (18:A-1–A11). As such, these primitives contain descriptions of import objects, export objects, attributes, and update functions. The executive primitives were constructed using these steps:

1. *Declare Primitive Object Classes* - The object classes depicted in Figure 7 became the object classes defined in the formal domain model of the application executive.
2. *Add Object Attributes* - The object attributes described by Appendix A were added to the formal model.
3. *Define Update Functions* - The information contained in the dynamic models defined in Appendix A served as the basis for the formal implementation of each primitive's update function. Specifically, the states and conditions expressed in the model formed the pre- and post-conditions used by the update functions to compute new values for their associated attributes.

4.4 Impact of New Executive Capabilities

During definition of the executive primitives and their update functions, it was necessary to consider the impacts of adding concurrent execution capability, a global clock, explicit control of data flows, and event handling to the Architect implementation of the OCU model.

4.4.1 Concurrency. The imposition of concurrency into an application complicates its execution. If Architect is to contain entities which execute concurrently, a number of difficult questions must be answered about the units of concurrency in the application:

- What is the level of concurrency? Will the application executive control primitives directly, or will the level of concurrency be at the subsystem level?
- How will each application subsystem and primitive synchronize its local time with the simulation clock? How does the clock advance? Will subsystems and primitives maintain local clocks?
- How does the executive manage the problem of concurrent producers and consumers of data?
- How do concurrent tasks synchronize their data in Architect? What happens to data that is produced faster than it can be consumed?

At what level do components execute concurrently in the Architect application executive? According to the OCU model, the application executive only "knows" about top-level subsystems. As far as the application executive is concerned, concurrency only occurs at the subsystem level. A domain may contain primitive objects that are designed to execute concurrently. An application specialist can force the executive to control primitive objects concurrently by modeling his application as a group of subsystems each controlling one primitive object. If an application specialist wishes to specify a model with concurrency below the subsystem level, he must understand the way primitive objects in the domain interact well enough to specify a method for controlling their concurrent execution.

4.4.2 Simulation Clock and Time. How does the application executive maintain simulated time? When does the clock advance? If the application specialist chooses a mode other than non-event-driven sequential, Architect automatically creates a global simulation clock. The Architect application executive uses the clock to keep absolute simulation time. Each subordinate component does not read the clock during execution. Each component raises events, in the event-driven mode, and stamps them with a *relative* time value. The event manager primitive keeps a copy of the current clock time in its import area, and it stamps the event with the correct absolute time before it stores it in its internal event list. This clock can only be updated to time $t + \tau$ (where τ is the relative difference in time between t and the scheduled time of the next event in the event manager's event list) when the following three conditions are true:

1. There are no more time t events scheduled.
2. It is not possible for any top-level subsystem to schedule any more time t events.
3. The next event in the application occurs at τ units after the current time.

In event-driven sequential and time-driven sequential modes, the application executive allows only one subsystem to execute at any time. The executing subsystem may request services from the executive that are serviced immediately, in the order in which they are received, before the clock increments. The application executive changes the clock time when all events at time t have been serviced. This conservative method of sequential execution is necessary to prevent causality errors among subsystems. This technique also prevents the executive from relying solely on the fact that the current implementation is on a sequential machine to guard against receiving events out of order.

4.4.3 Concurrent Data Synchronization and Connections. How does the application executive handle concurrent data synchronization and the producer-consumer problem? Sections 4.4.1 and 4.4.2 on concurrency and time discuss the way the executive will manage the flow of control. The executive must also manage the flow of data in the application. The application executive model in Chapter 3 assumes that all subsystems in the application are connected using a *connection* object. Connection objects isolate each component from the other components in the application. The application executive domain model assumes that each component can consist of other subcomponents. In the OCU sense, a component in the domain model can be either a subsystem or a primitive object. The domain model states that connection objects connect subsystems, at the subsystem level. Connections join primitive objects below the subsystem level, too. However, in the current implementation, subsystems and primitive objects below the subsystem level do not use connection objects. Instead, each import area looks to the export area of its source object or subsystem for input data. The previous implementation of Architect used this technique throughout each application. By permitting primitive import areas to contain information about their source exports, Architect allows these primitives access to some information about their external environment. Clearly, this approach violates the OCU premise that each primitive knows nothing about its neighbors or the way it is connected

to its neighbors (12:19). The KBSE group implemented top-level connections in the model during this research with the intent to isolate subsystems more fully.

There are three possible ways for concurrent subsystems to communicate, and these methods become important when one subsystem produces output at a faster rate than another subsystem consumes it. In one technique, the producing subsystem always overwrites what it produced, whether or not is was consumed. This forces the consumer to get only the most c' 'ent value. This method avoids the possibility of deadlock. Another technique directs the connection to queue all the data as it is produced. The consumer can consume all the data when it is ready and thus catch up. This method, too, avoids the possibility of deadlock. Finally, the producer can block on subsequent execution if its initial output is not consumed. Although this method may lead to deadlock, the application domain may call for this method to be used.

There are domains and applications where one method is preferred over another. Consider the following: an application models a barber shop. Process A is a customer arrival process, and Process B is a haircut process. If Process A outputs an arrival every five seconds, and a haircut takes fifteen seconds, then the second arrival will not get serviced unless the arrivals are saved in a queue. In this case, it is best to save all inputs to process B in a queue. Secondly, consider a missile with a radar system. The radar system updates the missile every five seconds, but the guidance algorithm takes fifteen seconds to complete. At the end of the fifteen seconds, the guidance system uses the most current data—it is the best data in this case. In other words, it is best to throw away all but the most recent data. Finally, suppose an application contains a controller that sends a command to a tank that controls tank pressure, and the tank sends the current pressure back to the controller. If the controller operates at a faster rate than the tank, and subsequent execution of the controller depends on the results of the tank, the controller must block to allow the tank to catch up and reach a stable state.

An application executive that can operate in all domains would allow the application specialist to specify which concurrent communication model he prefers. The Architect application executive, by virtue of only only allowing sequential execution, implements

the third method of controlling concurrent communication. The executive automatically blocks all processes but the consumer process, which consumes the data in the connection.

4.4.4 OCU-Specific Events and Delay Modeling. Implementation of the application executive together with the domain models developed by Waggoner (24) resulted in the addition of a number of application events to the event hierarchy depicted in Figure 7. These events are a by-product of Architect's OCU basis, which the application executive incorporates.

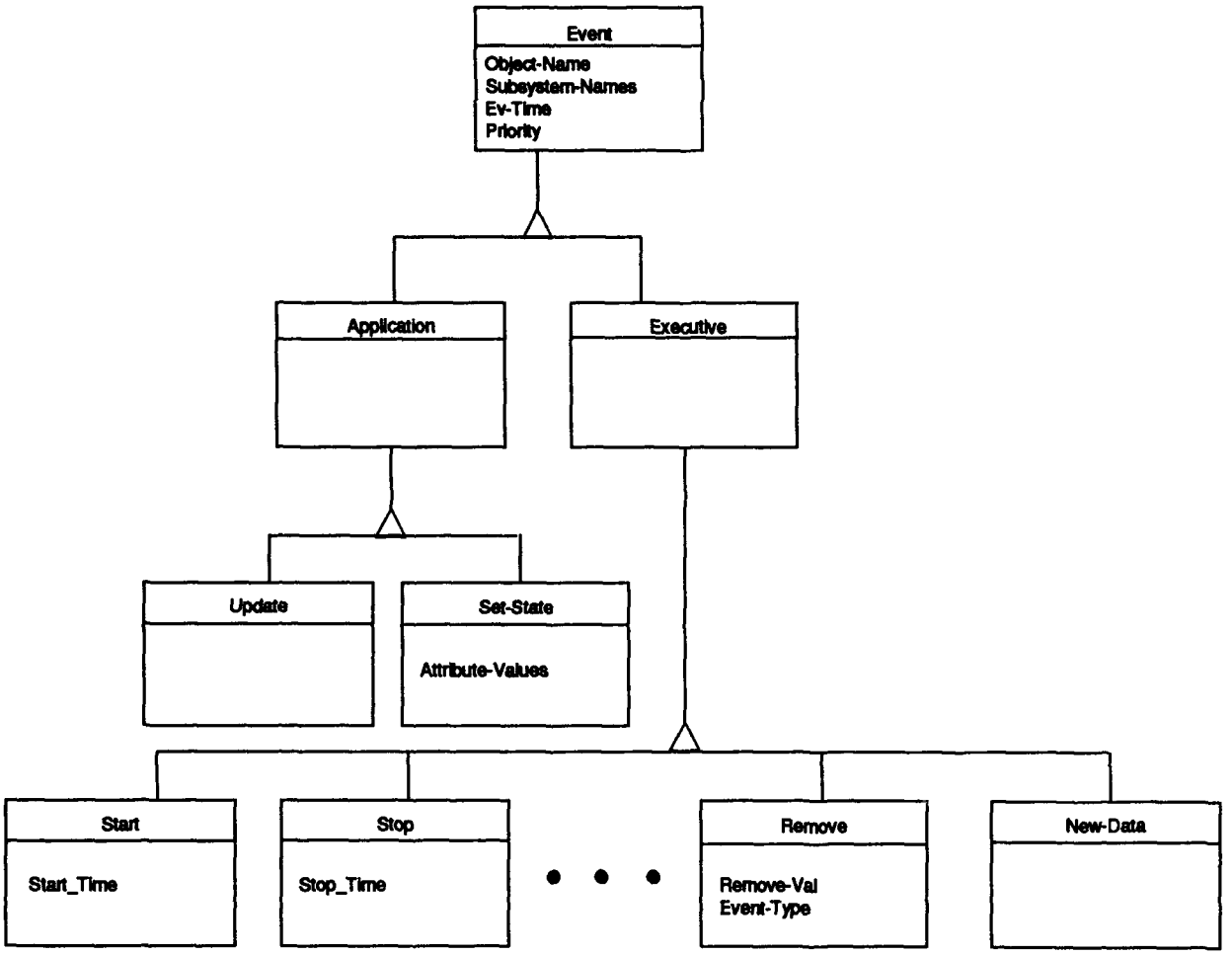

Figure 8. OCU Application-Specific Events

Figure 8 shows these specializations of the *application* event and *executive* event. These events are:

- *New-Data Event* - This event notifies a model subsystem that new data has arrived in its top-level subsystem import area. The subsystem responds by generating an update event for the primitive that consumes that data.
- *Update Event* - This event causes a primitive update function to execute.
- *Set-State Event* - This event executes a primitive's set-state function. A primitive schedules this event for itself when it wants to update its state at some future time.

The application executive uses *application* events such as *set-state* events to control when the state of a component changes with respect to time. Controlling the changing of state over time is an important concept because when a component executes at time t, its results depend on its state at time t. However, between the time the component is initially activated and the time when it produces its output, the component is in a transient state. If a component's output is queried during its transient state, invalid output can result. It is the domain and application specialist's job to define application events which deal with transient component states. The *set-state* event handles this problem. An application subsystem can change its internal state at time t and schedule a *set-state* event for τ time units later. The *set-state* event changes the component's output state at the same time the component broadcasts its output to the connection(s) it feeds using a *transmit* event, thus controlling the nature of this transient state. However, the order in which the *transmit* and *set-state* events are serviced is important. If the *transmit* event is serviced before the set state event, it may output the wrong data.

4.5 Transformation of Domain Model to OCU Structure

The application executive domain model specifies objects and operations which perform the executive services listed in Chapter 3 and detailed in Appendix A. The OCU architecture provides a useful method of encapsulating these objects and operations either into a set of related subsystems or a set of primitives, or some combination of both. This section explains the rationale behind the present application executive design.

Initially, it seemed appropriate to build one subsystem per application executive service, as Bailor suggested in (3). This method of encapsulating application executive

services has many advantages. It allows easy replacement of application executive services that operate in different modes. It permits the application executive designer to keep many of the same subsystems regardless of the application execution mode. The domain analysis in Appendix A pointed out that although each model component would behave differently in an event-driven application than in a time-driven application, the executive still services events when running a time-driven application. A time-driven executive services events as a result of changes in the clock. A time-driven application requests executive services using events that are serviced by the executive. An event manager subsystem would be useful in both the time-driven and event-driven modes. Similarly, a connection manager would be useful in all modes where top-level subsystems communicate with each other.

Assume that every subsystem in an application must operate using executive services. This would mean that an abstract executive composed of top-level subsystems that each perform a particular service would need to communicate using its own executive services. It would also be required to use its own services to keep track of each executive subsystem's state.

4.5.1 The Executive as Related, Top-Level Subsystems. The following is an example of how the event manager subsystem, the clock subsystem, and the component manager subsystem might interact to service a *start* event if the executive were implemented as sequential, top-level subsystems. Recall that the *start* event sets the global clock to the start time and signifies the beginning of execution. This scenario assumes that the application specialist has placed a *start* event on the event queue prior to execution. Although this example considers the event-driven mode of operation, it also applies to the time-driven mode.

1. Event Manager services *start* event.
2. Event Manager schedules a *transmit* event to send the start time to the Clock Manager.
3. Connection Manager services *transmit* event for itself, to transmit starting clock time contained in the event.
4. Connection Manager gets start time from Event Manager's export area and writes it to the connection object which links the event manager and the clock.

5. Connection Manager schedules a *receive* event for the connection to the Clock Manager.
6. Event Manager services *receive* event for the connection to the Clock Manager.
7. Connection Manager writes the data to the Clock Manager.
8. Connection Manager schedules *update* event for Clock Manager.
9. Event Manager services *update* event for Clock Manager.
10. The Clock Manager's controller updates the subordinate clock object primitive. Now the clock object's export area contains the new clock time which may be transmitted to event manager using a sequence of *transmit* and *receive* events not depicted here.
11. The Event Manager has now serviced the *start* event.

4.5.2 The Executive as a Single, Top-Level Subsystem. Currently, the Architect application executive consists of a different application executive subsystem for each mode of operation. This approach requires the use of a specialized executive subsystem controller that routes *update* events scheduled by executive primitives for executive primitives. The following scenario illustrates the interaction between primitives for an event-driven concurrent application executive as it services a *start* event.

1. Executive Controller updates Event Manager.
2. Event Manager services *start* event.
3. Event Manager sets primitive export with proposed start time.
4. Clock Manager import is updated.
5. Event Manager schedules *update* event for Clock Manager.
6. Event Manager passes this *update* event up to the Executive Controller.
7. Executive Controller routes *update* event to Clock Manager.
8. Clock Manager sets itself to the current time, completing service of the *start* event.
9. Executive Controller updates Event Manager again.

The example scenarios listed above illustrate the major disadvantage of building an application executive as a collection of related subsystems—complexity. The number of events generated by the set of executive subsystems for each other, far outstrips the number of events generated by a single executive subsystem that contains primitives that provide executive services. This complexity also obscures which events control the application and which events control the executive. Both *update* for the application and *update* events for

the executive are placed on the executive's event list. If Architect's application executive is implemented as a single subsystem, the events which control the executive could be passed "up" to the subsystem controller, while the application service events (*transmit* and *receive* events for the model components) flow "across" the primitives.

Analysis of the way application executive objects interact during the servicing of one event reveals their close cooperation. For example, in event-driven sequential mode, when the event manager services an application event, it calls the subordinate application component update function. It collects events from the subsystem and schedules them. As a result of event service, the executive may schedule update events for the connection manager (when servicing *transmit* and *receive* events) or for the clock manager (as shown in the above scenario) This close interaction between executive objects is another reason why the Architect application executive is implemented as a single subsystem that encapsulates its services in a group of related primitives. Specifically, an application executive controls some or all of these primitives:

- Component Manager Primitive - This primitive keeps the execution state of all application subsystems by keeping set a of object control blocks (OCBs). This primitive contains the method necessary to maintain the set as well as modify the attributes of the individual elements.
- Device Manager Primitive - This primitive keeps the execution state for all input/output devices used by the application using a set of device control blocks. This primitive contains set maintenance methods similar to the Component Manager Primitive.
- Event Manager Primitive - This primitive handles all events for the application. This primitive contains an update method that adds events to an event list, services the event, and removes the event from the event list. This primitive employs REFINE function calls that pass control to model components and collect events raised by the components during execution.
- Connection Manager Primitive - The Connection Manager Primitive contains a set of connection objects that link subsystems in the application. This primitive contains

update methods that return connection state when needed. The methods also read data from and write data to component import and export areas directly. This primitive raises events that must be placed on the event list by the event manager.

- Clock Manager Primitive - This primitive keeps time for the application.

The domain model presented in Chapter 3 contains a few more objects than those mentioned above. The registrar object is absent from the application executive subsystem. The dynamic model drew a clear line between creation of the application executive and the execution of the executive. Similarly, Architect draws a line between composition and execution. The application specialist composes an application by interacting with the Architect visual system (5). After the application specialist composes the application, Architect executes the application using the predefined application executive. Thus registration of model components is carried out by Architect during and immediately following composition. Therefore, the application executive does not contain the registrar explicitly as a primitive object, but the registrar is included implicitly as a function which is part of Architect. The device object and component object depicted in the application executive object model are part of the composed model and not part of the executive. These objects are constructed by the domain engineer as part of domain analysis. They are brought into the application by the application specialist during model composition. The executive controls them like other subsystems. Since they are also not application executive objects, they are not in the application executive subsystem.

4.6 Conclusion

The informal domain model was transformed into formal domain model primitives which conform to the Architect paradigm. These domain model primitives encapsulate object attributes and methods. Chapter 5 describes how these primitives are combined to form a set of single subsystems that act as the Architect application executive.

5. *Executive Domain Model Instantiation*

5.1 Introduction

The set of domain model primitives expressed as a collection of REFINE object, attribute and function declarations describes the services an executive subsystem provides. The primitives do not constitute an executive unless they are joined together under the control of a subsystem controller and they become an Architect subsystem. This chapter examines the process of joining these primitives together. It describes the results of composing the primitives to form an event-driven sequential and a time-driven sequential executive. Although there are no concurrent executive subsystems in Architect, this section explains how one may compose executive domain model primitives to form event-driven concurrent and time-driven concurrent executives.

5.2 Instantiation Technique

When an application specialist composes an application in Architect, he or she uses the visual system interface. The application specialist composes icons representing primitives into graphical representations of subsystems. In the domain of application executives, there are no icons associated with each of the primitives. This is because when Architect is in normal use, the application specialist does not see the application executive subsystem. The application executive subsystem is brought into the current application automatically and is invisible to the user. The lack of icons for each executive primitive prohibited using Architect's visual system to compose the application executive primitives into a subsystem. Each application executive subsystem was instantiated in the following manner:

1. Determine which executive services are required for each of the four modes of executive operation.
2. List the corresponding primitives that perform these services whether separately or together.
3. List the primitive interface types, as defined by the data-type and data-category fields in their respective import and export areas.

4. Define a file for each proposed executive subsystem containing definitions of the primitives required for that particular mode, the import and export areas for each of these primitives, and the subsystem that controls each of these primitives.

These subsystems were defined by parsing in a file containing OCU and application executive domain-specific grammar. However, except for the lack of icons, there is no reason why the visual interface could not have been used to instantiate these models. The following sections detail the results of using the instantiation method listed in this section.

5.3 Concept of Operations

The primitives described in Section 4.5 work together with the application subsystems. They provide a number of services commensurate with the mode of application created by the application specialist. The application executive primitives are created and connected to the model they are controlling. This section discusses the two phases of executive operation: registration and execution. During the discussion of the execution phase, it gives the results of the instantiation method described in Section 5.2.

5.3.1 Registration Phase. The application specialist composes the desired model. He selects the desired mode of operation: event-driven sequential, time-driven sequential, or non-event-driven sequential. Architect automatically reads a file containing a pre-defined executive subsystem and places it in the current application specification abstract syntax tree. The application specialist then defines the control routine for the application. The format of the control routine depends on which mode the application specialist chooses. In the case of an event-driven application, he or she edits the list of events which will occur during execution. In the non-event-driven sequential mode, he or she specifies an application update function similar to the update functions defined by (2, 18). In the time-driven sequential mode, the application specialist defines an event for each subsystem. Then the executive forces the model to respond to changes in the system clock.

5.3.2 Execution Phase. Following registration, the application is ready to execute. The sections below describe which executive primitives are required during each mode and how the primitives interact to manage flow of data and control in the application.

5.3.2.1 Non-Event-Driven Mode. This mode requires only one executive service — control of subsystem execution order. Architect operates on a list of subsystem update calls in the manner described in (2, 18). This service is implemented by executing a list of update calls contained in an application's application object. Because the executive function in non-event-driven sequential mode is handled by the application object, the application executive does not exist as a separate subsystem. Architect runs the application by reading and executing each application subsystem update statement in the *Application* object sequentially. This method of operation is identical to the way Architect worked as described by Anderson and Randour (2, 18). It does not include the use of connection objects at any level in the composed model. The KBSE group included this mode of operation to keep Architect backward-compatible enough to allow group members to use the baseline established during previous research to continue with research aimed specifically at extending Architect's technology base. (See Warner (25) for more details on the use of this execution mode and on technology base extension.)

5.3.2.2 Event-Driven Concurrent Mode. An executive running an application in event-driven concurrent mode requires the following primitives:

- Event Manager
- Connection Manager
- Component Manager
- Device Manager - if there are any I/O devices in the model
- Clock

Although Architect cannot currently execute an event-driven concurrent application, this section explains how the application executive subsystem might work during execution in the event-driven concurrent mode.

The following is an example of how the event manager primitive, the clock primitive, the connection manager primitive, and the component manager primitive interact to service an *update* event. This scenario assumes that the application specialist has placed a *update* event on the event queue. Although this example considers the event-driven concurrent mode of operation, it also applies to the time-driven concurrent mode.

1. Event Manager primitive's *current-event* attribute contains an *update* event for component A.
2. The Event Manager sets its export with the value in the update event for use by other primitives.
3. The Event Manager primitive passes control up to the executive controller as it schedules an *update* event for the connection manager, the component manager, and itself.
4. The executive controller updates the connection manager primitive.
5. The connection manager primitive checks the status of Component A's downstream connection and makes sure it is *Consumed*.
6. The connection manager update terminates and returns control to the executive controller.
7. The executive controller updates the component manager primitive.
8. The component manager primitive changes the state of component A's object control block based on this request to execute and its current state.
9. The component manager passes control back to the executive controller.
10. The executive controller updates the event manager so it can now, armed with the necessary component and connection status information, service the current event (which is the original *update* event).
11. The event manager services the *update* event depending on the component and connection status information.
12. If the proper conditions exist to pass control to the component, the event manager calls an Architect function that passes the *update* event down to the subordinate component and allows it to execute.
13. If the event cannot be serviced at this time, it is placed on the event manager's suspended list.
14. The event manager deletes the *update* event from the event list, revealing a new current event.
15. This cycle repeats itself until the event manager services a *stop* event.

5.3.2.3 Time-Driven Concurrent Mode. At present, Architect's application executive cannot run applications in a time-driven concurrent mode. If Architect could run models in this mode, it would need the following executive primitives:

- Event Manager
- Connection Manager
- Component Manager
- Device Manager - if there are any I/O devices in the model
- Clock

These are the same primitives required in event-driven concurrent mode. The only difference between these two modes of operation is the way the event manager operates. In time-driven concurrent mode, the event manager contains a list of application events which occur during one clock "tick." (A "tick" is a change in the global absolute time.) The executive simulates concurrency by allowing more than one component to execute at a given time *t*. When all the events on the event list have been serviced, the event manager causes the clock to tick and re-services the application events processed during the previous "tick." The event manager removes the executive events (e.g. *transmit*, *receive*, *activate*) that the application primitives and executive primitives schedule during execution. The model components may also add application events to or take application events away from the list of events the event manager services each time the clock changes. This capability allows the application specialist the freedom from forcing his model components to execute every clock tick. The executive primitives communicate in time-driven concurrent mode in the same way they communicate in event-driven concurrent mode.

5.3.2.4 Event-Driven Sequential Mode. An application specialist can currently execute event-driven sequential applications using Architect. This mode's executive subsystem consists of:

- Event Manager
- Connection Manager

- Clock Manager

These primitives are sufficient because of the restrictions this mode of operation places upon the application. In this mode, no two subsystems are permitted to execute at any one time. Sequential execution precludes the possibility of one subsystem producing data faster than it can be consumed. No component manager is needed to monitor the execution state of each application subsystem. Similarly, no device manager is required to monitor device status, because no device will put data into or take data from the application at a rate greater than the application executive will allow. The application specialist must schedule *update* events for input devices. When an output device receives new data, it will schedule an *update* event for itself. Application specialists must schedule at least one *update* event to begin application execution.

5.3.2.5 Time-Driven Sequential Mode. Architect can execute time-driven sequential applications. A time-driven sequential application's executive subsystem controls these primitives:

- Event Manager
- Connection Manager
- Clock Manager

This small subset of executive primitives is sufficient for operating time-driven sequential applications. Time-driven sequential mode requires the executive to enforce the same constraints on execution as an executive in event-driven sequential mode (see Section 5.3.2.4 above).

The domain model described in Appendix A discusses how a time-driven executive also uses events to control communication between and activation of subsystems. The time-driven sequential application executive operates in the same manner as the event-driven sequential application executive with two exceptions: event management and clock management. In time-driven sequential mode, time is provided, via a connection object, for all primitives in the application that request it during registration. In the time-driven sequential mode, the event manager does not delete application events from the event list

unless explicitly told to do so by the application. (An application notifies the executive to remove the application event by raising a *remove* event.) Instead, the event manager services each *application* event in a round-robin fashion until the clock is at the value contained in the *stop-time* attribute of the *stop* event in the event list. The event manager deletes *transmit* and *receive* events after the event manager and the other primitives complete servicing them. Thus, in time-driven sequential mode, all applications must contain a stop event or they will not terminate. Figure 9 shows how primitives are related in an event-driven and time-driven application executive. The figure shows the only difference between the time-driven sequential and event-driven sequential application executive — in the time-driven executive, the clock creates a *transmit* event to broadcast the current time to the application subsystems.

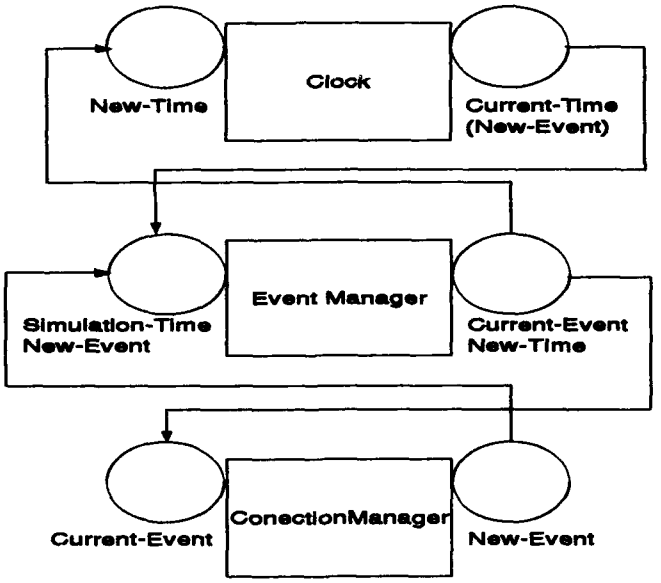

Figure 9. An Application Executive for any Sequential Mode Application

5.4 Implementation Technique and Additions to Architect

During development of the two application executive subsystems, it was necessary to augment the previous implementation of Architect. Addition of an executive required changes to the following areas:

1. The OCU grammar and OCU domain model
2. The Architect system software
3. The Architect Visual System Interface (AVSI)

5.4.1 OCU Domain Model and Domain-Specific Language. According to Lee, an application executive is a specialized subsystem that controls OCU model execution (12). As such, the application executive is a part of the OCU domain model. Addition of the application executive to Architect during this thesis cycle resulted in the addition of the following artifacts to the OCU domain model:

- Event Objects
- Connection Objects
- A Descriptor Object
- Executive Primitives

During composition, Architect's visual system creates an abstract syntax tree rooted at a node called a Spec-Obj. As the application specialist includes items from the technology base in the current application, Architect adds children (Spec-Parts) to this root node. The previous implementation contained application objects, subsystem objects, and primitive objects as possible Spec-Obj children. The current implementation of Architect contains an additional Spec-Part: A Descriptor-Obj.

The Descriptor-Obj serves two purposes. It contains an attribute which describes the mode of the application. Second, it provides a place for Architect to store connection objects as the application specialist creates them. When Architect loads the predefined application executive, it copies the set of connection objects into the executive's connection list. The executive manipulates the connection list during execution.

The method of including pre-defined application executive subsystems in each composed application requires inclusion of application executive domain-specific grammar in the OCU architecture-specific language (ASL) grammar. Currently, Architect uses this grammar to parse executive subsystem definitions into the REFINE object base from a file.

Gool (8) discusses changes to the Architect implementation of the OCU model during this research.

5.4.2 Architect System Software. Each application subsystem in Architect uses the same Architect system software to drive its controller. This routine gathers the events passed to it by the executive and routes them to their respective primitives. Then, this router collects them and passes them to the executive for inclusion into the event manager primitive's event list. The executive subsystem's controller works differently.

As a specialized subsystem, the application executive requires a special Architect system software routine to run it. This function, called *Execute-Exec-Subsystem*, routes events raised by the application executive primitives for other primitives during execution. It accounts for the fact that in order to service an event, many executive primitives must cooperate before servicing the next event by not updating the event manager until all other executive primitives finish servicing the current event, as output by the event manager. This routine operates in a manner similar to Gool's *Execute-Subsystem* (8). It accepts events from subordinate primitives, and it acts as a control routine for the executive subsystem by re-directing these events to the appropriate executive subsystem. *Execute-Exec-Subsystem* operates differently from Gool's function by not passing events up to the calling routine. *Execute-Exec-Subsystem* operation conforms to the OCU idea that the executive should act as a specialized subsystem by controlling the executive subsystem in a different way than other model subsystems.

5.4.3 Architect Visual System. The application executive requires the application specialist to enter execution information while composing an application. The user must enter the application mode and depending on the mode, must define an initial set of events or update statements. Cossentine modified the Architect Visual System Interface (AVSI) to allow the user the opportunity to perform these activities (5). These modifications consisted of additional menus and prompts that the user interacts with prior to execution. For example, Architect parses the executive subsystem from a Unix file and initializes connection objects during these interactions with the user.

5.5 Conclusion

Architect's application executive consists of a single subsystem that provides the executive services derived from the domain model documented in Appendix A and described in Chapter 3. Appendix C contains the application executive subsystems for event-driven sequential and the time-driven sequential execution modes. Other executive subsystems that control time-driven concurrent and event-driven concurrent applications can be instantiated using the method and suggestions given in this chapter. Chapter 6 outlines the test cases and procedures used to test the two application executive subsystems.

6. *Architect Executive Validation and Analysis*

6.1 Introduction

Chapter 5 described instantiations of the Architect application executive subsystems in each mode of execution. This chapter explains the method used to verify that the application executive performs correctly in event-driven sequential and time-driven sequential mode. It describes the applications used to test each application executive subsystem. It lists the expectations and the results of each test. It shows why these test are sufficient to demonstrate correct operation of the Architect application executive.

6.2 Validating Domains

Small applications in the domains of logic circuits and cruise missiles validated the operation of the application executive in the event-driven sequential and time-driven sequential modes. Waggoner constructed the logic circuit domain primitives used in the event-driven sequential executive test. Waggoner's logic circuit primitives are similar to those developed by Anderson and Randour with two exceptions: they generate events and they use their delay attributes to generate events for future times. A detailed discussion of all primitives in the event-driven logic circuit domain can be found in (24). These primitives made up the small logic circuit used in the test:

- *And-Gate-Obj* - Outputs one result of a logical and of two input signals.
- *Not-Gate-Obj* - Outputs a single negated input signal.
- *Switch-Obj* - Generates a logic one or logic zero.
- *LED-Obj* - Acts as a terminator object. Outputs its state to the REFINE interaction window when its state changes.

Waggoner also constructed the cruise missile domain primitives used in the time-driven sequential application executive test. A detailed description of these primitives is contained in (24). These primitives combined to form two simple cruise missile applications:

- *Fuel-Tank-Obj* - This primitive simulates a cruise missile fuel tank. It models a pump that pumps fuel at a predefined rate when the pump motor is set to *on*.

- *Throttle-Obj* - This models an engine throttle. It contains a throttle setting attribute, *Max-Flow-Rate* which is designed to show what percent of the greatest possible fuel flow the engine should get.
- *Jet-Engine-Obj* - Based on whether or not the *start* signal is asserted, the engine's *mode* is *off*, *starting*, or *running*. This primitive consumes fuel and outputs thrust.
- *Start-Switch-Obj* - The *Start-Switch-Obj* supplies a signal to cruise missile primitives. In the applications created here, this primitive supplied an input signal to the *Jet-Engine* and *Fuel-Tank* primitives.

6.3 Testing Technique

As with the executive instantiation phase of this research, application executive testing relied upon use of the Architect textual interface developed by Anderson and Randour (2, 18). It was necessary to use the textual interface during executive testing because modification of the AVSI to support the executive occurred concurrently during this research. Relying on the textual interface forced the test results to remain focused on errors introduced by the executive and not on possible errors introduced by the interface system. The textual interface allowed each test application to be precisely defined prior to running each test case, thus allowing complete control over the contents of the REFINE object base during the test.

A text editor constructed the test case applications and the Architect *Parse-File* utility converted the application definition, complete with its fully-defined application executive subsystem into objects in the REFINE object base. Then the Architect Execute-Application function exercised the application executive in these three test cases in both the time-driven sequential mode and the event-driven sequential mode:

- Correct operation of the executive subsystem alone. This test looked at the executive's ability to correctly service *start*, *stop*, and *transmit* events.
- Correct operation of the executive subsystem when controlling one subsystem. The second test's purpose was to show that the executive could correctly service *transmit*,

receive, *new-data*, *set-state*, and *update* events and accept events from the subsystems it controls.

- Correct operation of the executive subsystem when controlling two subsystems. This test looked at correct service of all events and correct use of connection objects as well as correct data flow throughout the application.

What does "correct operation" really mean? In Architect, the following criteria help one determine whether or not the executive functioned properly:

- The clock increments when the event manager has serviced all time t events and it is not possible for any primitive to raise more time t events.
- The event manager and connection manager schedule the correct number and type of events in response to the current event.
- The current event changes when the event manager and the other primitives have all completed their contribution to servicing the current event.

A list of initial events constituting part of a completely defined application executive definition were parsed in prior to running each test. The state of each subsystem was observed before running the test. During execution, the event manager primitive serviced each event and the application displayed messages concerning the application's progress in the REFINE interaction window. As events completed service, the Event Manager appended them to a list of old events. After execution, all events that made it into the event manager's event list and were serviced were in this old event list in the order of service. Inspection of the old event list and the state of each subsystem and primitive demonstrated that not only were events serviced in the predicted order, but the test application components raised the correct events in the proper order in response to the events on the event list in the beginning of the simulation.

6.4 Specific Test Cases and Results

This section contains a description of each test case and a summary of the results of each test, grouped according to the mode of the test application.

6.4.1 Event-Driven Sequential Test. Appendix B contains representative test cases for the event-driven sequential test and a sample of the test results. Figure 10 shows the simple application that tested the executive's ability to control model subsystems in this mode of execution. Figure 11 depicts the two subsystem application that exercised the executive's data flow control capability.

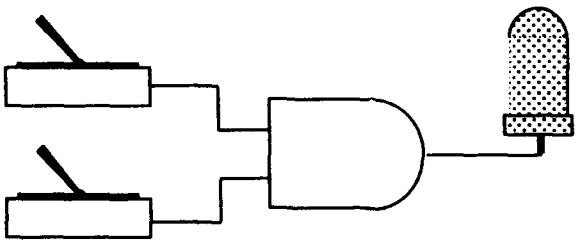

Figure 10. A Circuit Application With One Subsystem Tested the Event-Driven Sequential Application Executive Subsystem

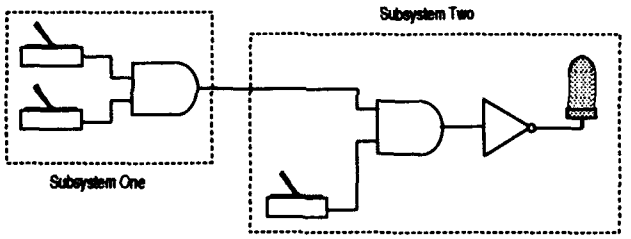

Figure 11. A Circuit Application With Two Subsystems Used to Test the Event-Driven Sequential Application Executive Subsystem

The event-driven executive tests revealed that the application executive correctly serviced all application and executive events. In the case of the multiple subsystem test, the connection object changed state accurately in response to the consecutive service of *transmit* and *receive* events. The one subsystem and two-subsystem tests correctly modeled the delay through the circuit. In the one subsystem test, the *LED* changed to the *on* state after a five second delay. In the two subsystem test, the *LED* object transitioned to the *off* state after a delay of fifteen seconds after one of the input switches changes position during model execution.

6.4.2 Time-Driven Sequential Test. Appendix B contains representative test cases for the time-driven sequential test and a sample of the test results.

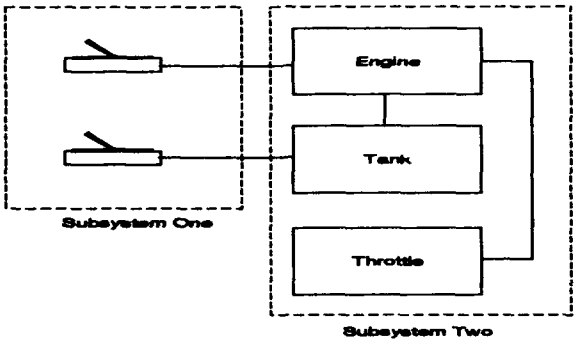

Figure 12. The Multiple Subsystem Cruise Missile Test Application

The cruise missile domain primitives used in the time-driven sequential tests were considerably more complex than the logic circuits domain primitives in the event-driven sequential domain tests. Figure 12 shows the two subsystem test configuration. The one subsystem test consisted of the subsystem with an engine, tank, and throttle where the engine start and fuel pump start inputs set to *on* (true). However, control of the primitives was simpler in one respect: the primitives did not schedule events for themselves. Instead, these tests focused on whether or not the correct time value was broadcast to each primitive that required time. This broadcast had to take place before each of these primitives serviced their pre-scheduled update events, as they had to use a comparison of the current time to the previous time when setting their state attributes.

The tests showed that time was transmitted to and received by each primitive prior to each primitive's update function. Also, the primitives changed state appropriately. During each simulation, the engine primitive transitioned to the *Running* state and it generated thrust. The executive primitives cooperated correctly as they incremented time, managed communication, and serviced events throughout the application.

6.5 Performance Analysis

This research was not aimed at optimizing application executive performance and Architect execution. However, a few words can be said about the impact of the application

executive on Architect performance. The effects of the application executive on Architect execution performance are due to the need for executive primitives to be isolated from each other in order to comply with the OCU model and the fact the executive calls Architect system routines during its operation.

The separation between the event manager and the connection manager led to an inefficient technique of handling *transmit* events. Each time an application primitive called *Set-Export*, the primitive generated a *transmit* event which was later placed on the *event-list*. The connection manager respond to each *transmit* event whether or not a *connection* object existed for the export being set. This inefficient method is a by-product of the need for the connection manager to keep the secrets of each *connection* object, and the need for each primitive and subsystem to not contain knowledge about any primitive or subsystem outside its locus of control, including primitives in the application executive subsystem.

The application executive makes many calls to the Architect system routines *Set-Export* and *Get-Import* in order for it to operate. These system calls slow system performance. For example, during the time-driven tests, it was assumed that when the event manager was doing nothing but incrementing the clock and broadcasting time to the application subsystems that the system would run fairly fast. Instead, the clock update message appeared in the REFINE interaction window at a rate of one message per second. This was due to the volume of events required to broadcast the time as well at the large number of calls that the executive primitives needed to make to service these events.

6.6 Summary

Testing of the Architect event-driven sequential and time-driven sequential application executives revealed their correct operation. Correct operation is defined by the executive's proper response to events in the event manager's event list. The following chapter, Chapter 7 views the test results presented here in the context of the sum of this research.

7. *Conclusions and Recommendations*

7.1 Introduction

This chapter summarizes this research's goals and compares them to its accomplishments based upon the results presented in Chapter 6. It discusses the effectiveness of the domain analysis process, draws conclusions based on the results, and makes suggestions for further research.

7.2 Research Accomplishments

Recall the purpose of this research:

> ***The purpose of this research was to evaluate, specify, implement and analyze an application executive subsystem model which allocates host machine and subsystem resources during application execution.***

This effort resulted in these accomplishments:

1. *Augmented OCU Model* - This research determined extra constructs needed for the OCU model. Prior to this research the OCU model lacked elements needed to implement a domain model containing the common elements identified above.
2. *Constructed Informal Domain Model* - This research's informal domain analysis resulted in an informal domain model expressed in Rumbaugh object, dynamic, and functional models.
3. *Built Formal Domain Models* - This research augmented the application executive function of the previous version of the Architect system using instantiated application executive primitives. Requirements for these application executive subsystems included those identified in (2, 18) as well as those identified in a study of the OCU model and its implementation in Architect.
4. *Validated Executive Operation* - This effort validated operation of the executive in the event-driven sequential and time-driven sequential modes of operation. Event-driven sequential applications were built using domain model primitives from the domain of logic circuits. Time-driven sequential applications made from primitives in the cruise missile domain.

7.3 Architect Executive Capabilities

This research augmented the application executive of the previous version of Architect to allow the executive to explicitly control data flow, manage flow of control, and to model simulated time and delays during execution. This research created an application executive domain model that allows applications to execute both a time-driven and event-driven manner. These changes permit the Architect to function over more domains than the previous version. This research consisted of two phases, domain analysis and primitive instantiation. The following sections discuss the effectiveness of the methods used to accomplish each phase.

7.3.1 Application Executive Domain Analysis. This research created a flexible, adaptable domain model of an application executive that can be used by Architect to control application execution. The creation of the application executive resulted from a domain analysis over the domain of supervisory programs. The domain analysis process combined the functional analysis techniques of Tracz with the object-oriented analysis techniques of Prieto-Díaz. Initially, the research resulted in an informal model expressed in Rumbaugh diagram notation and text. These diagrams and textual descriptions were useful during domain model implementation, but they were not unambiguous enough to take advantage of REFINE's predicate transformation capability by themselves. Therefore, the informal models were transformed into formal REFINE object and function declarations using a set of heuristics (See Chapter 5). Ideally, it should be possible to initially define a domain model in mathematical terms, write this description in a text file, and parse it into the REFINE object base. Functions written in REFINE could transform these mathematical structures into OCU structures. The goal of creating a formal mathematical model prior to implementing a formal model as a set of Architect-OCU primitives was put aside to allow time to complete implementing the executive subsystems themselves.

The domain analysis process developed here correctly identified the iterative nature of domain analysis. During the *Abstract Objects* phase of domain analysis, the relationships expressed in the dynamic models changed as the primitives were developed. For example, as the more was understood about the need for the *Time-Driven Clock* object to broadcast

the current time to subordinate primitives, its export area changed to include a *New-Event* export. Development of the dynamic models took an inordinate amount of time. This was due to the complexity of the event traces, as well as the need to map the activities in the textual event traces into graphical representations. The resulting diagrams were useful, but perhaps a different dynamic modeling technique (one that takes only one "step" but still contains the same information) should be used during future domain analysis.

7.3.2 Application Executive Instantiation. Instantiating a particular application executive was relatively straightforward. This is because the ways in which the primitives could interact was determined during domain analysis. Instantiation became a process where primitives were plugged together. As these primitives were plugged together, it was discovered that one primitive, the connection manager, could be used as is in both the time-driven and event-driven application executives. The clock manager required a slight modification between the two execution modes, while the event manager required extensive modification. This primitive's update function was changed extensively to convert it from an event-driven sequential primitive to a time-driven sequential primitive.

7.4 Utility of OCU Structure

Many strengths of the application executive result from encapsulating it in an OCU structure. The presence of a common procedural interface to application subsystems permits the executive to interact with each subsystem in a similar way. This allows the executive subsystem to remain independent of the implementation of each application subsystem. This quality limits the amount of information that the executive is required to process during its registration phase. The use of a common procedural interface also allows the executive to drive any number of subsystems. The structure of the OCU model determined how the formal domain model components would be implemented. It drove the executive into an implementation which relied on the use of OCU primitives. The OCU structure simplified design trades between an executive that is implemented as a group of subsystems or a group of primitives which are united being under the control of a single executive in that it provided a starting point for the trades. Without this starting point,

if the executive was implemented as a series of functions written in REFINE, then this set of functions could be written with no organization save a functional one. The method of encapsulating the executive in a group of related primitives will enable future users of Architect to understand the types of services provided by each application executive primitive, and to extend those services incrementally. They will be able to substitute one primitive implementation for another. They will be able to combine the primitives used in the event-driven and time-driven test cases along with additional primitives to build executives which execute applications in the concurrent modes.

7.5 Suggestions for Further Research

Further research in the area of application executive definition should concentrate on the following areas:

1. *Visualize application executive* - Section 7.3.2 described the straightforward nature of connecting application executive primitives to form an instantiated executive subsystem. Future research should include definition of icons for the executive primitives to make it possible to use AVSI to instantiate executive subsystems.
2. *Refine application executive* - Architect is really only as powerful as the types of applications it can create and execute. While it can be used to compose an application from many different domains, it can only execute three types of applications, non-event-driven sequential, event-driven sequential and time-driven sequential. It cannot control concurrent applications. The informal domain model presents details on how a concurrent executive might work. The informal domain model artifacts should be transformed into primitives and instantiated using the methods outlined in this research. This concurrent capability may lead to a real-time application executive.
3. *Define more application domains* - During the validation of the application executive, the characteristics that made certain primitive event-driven, and certain primitives time-driven became clearer. Event-driven primitives schedule requests for self-activation with the executive, as well as requests for executive services. Time-driven primitives, on the other hand, do not schedule *update* events for themselves but may

request all other executive services by scheduling *executive* events with the executive. Armed with this information, future researchers should create more application domains to further test these notions for what it means for a primitive to be time-driven versus what it means for a primitive to be event-driven.

4. *Test and build more applications* - Testing only shows the presence of errors. Although Waggoner tested the application executive extensively using a number of applications constructed from the event-driven circuit and cruise missile domains (24), more applications should be created to further test executive operation.

5. *Investigate visualizing execution* - The executive allows the application specialist to view the events that the executive services and the time the executive services them. This permits a more detailed understanding of the temporal execution state of the application. The presence of a global time primitive allows future Architect developers to extend visualization of the application to include visualization of the passage of time. Future research should included an analysis of the utility of visualizing domain model execution.

7.6 Final Comments

This research successfully developed an executable domain model of an application executive for Architect. The informal domain model and formalization technique developed and demonstrated during this research provides a means for increasing Architect capabilities through continued improvements to the Architect application executive.

Appendix A. Application Executive Domain Model

A.1 Introduction

This appendix documents the development of an application executive domain model. It follows the domain modeling process defined in Chapter 3. The first six sections discuss the informal phases of domain analysis. The last section shows the results of formalization of the domain model using the informal artifacts derived in the early sections. It should be noted that during domain analysis, many of the steps in the domain analysis process were re-visited throughout the research. The informal and formal artifacts depicted here reflect the final results of domain analysis over the domain of application executives.

A.2 Name Domain

This research was undertaken by the AFIT Knowledge-Based Software Engineering (KBSE) group to improve its application composition system called Architect. One of the improvements that was called for was extending Architect's application execution capabilities (15:8). As a result, the domain of interest for this domain analysis process became the domain of software system executives.

A.3 Scope Domain

The domain of system executives includes programs that control the execution of other programs. Typically, operating systems are process-oriented and a process is viewed as a program in execution (1). Architect's application executive is similar to general operating system kernels in that it controls the execution of its subordinate entities. However, the application executive is not process-oriented. It fits into Architect's OCU architecture and manages the execution of subsystems instead of processes. The application executive controls subsystem communication by controlling links between subsystem import and export areas. Architect's application executive runs on top of a host operating system, so it need not be concerned with memory management or file management per se, but it may make calls to the host operating system to store and retrieve information during execution. Besides the architectural restrictions on the application executive imposed by

the Architect implementation of the OCU model, the research goals outlined in Chapter 1 impose other constraints. For example, the Architect application executive does not meet hard real-time constraints. This application executive is not designed to run on a parallel machine. Although the enhancement of Architect's user interface is beyond the scope of this research, the application executive model includes a means for the user to interact with the composed application and understand its behavior as it executes. Chapter 1 also specifies that Architect's application executive must:

- Control the use of all host system interfaces including the CPU and I/O resources
- Permit concurrent subsystem execution
- Maintain a system clock
- Manage subsystem execution time
- Allow sharing of data between subsystems

A complete domain model of an Architect application executive includes objects and operations to perform the functions described above.

A.4 Obtain Domain Knowledge

Given the limited scope of the domain as outlined in section A.3, there are key operating systems and OCU modeling concepts the Architect application executive employs. These concepts include OCU architectural considerations, the services provided by the J-MASS executive, and constraints implied by a need to potentially control concurrent execution. These ideas allow a domain analyst to identify the objects and operations that make up an application executive domain model. This section describes these concepts and defines a set of executive services that the application executive domain model describes.

A.4.1 The OCU Architecture. The OCU model defines the framework for the Architect application executive. Chapter 2 states an application that is modeled in the OCU paradigm consists primarily of subsystems. A subsystem operates upon a group of objects and these objects change state based upon the data presented to the subsystem import area and the subsystem procedure called by the application executive (12:18-19).

In order for an application executive to comply with the OCU model, it uses the subsystem procedural interface to cause subordinate subsystems to execute. The application executive model itself conforms to the OCU model definition. Just as the subsystem is "the locus of a mission and the objects are services to carry out the mission" (12:18) the application executive serves as the locus of the application's mission. The OCU model provides for an interface with host system resources in the form of an I/O driver, a control surrogate, and a monitor surrogate (12:23). These constructs help the executive communicate with resources external to the application. The monitor surrogate communicates with the application and the control surrogate communicates with the host hardware. The application executive developed during this research does not use the structures listed above (monitors and surrogates) to implement these services. It does incorporate information hiding and resource protection in the application executive structures that take the place of monitors and surrogates. All of these OCU structures have an impact on the kind of objects in the application executive domain model.

D'Ipollito hinted at the services an OCU-compliant software application's executive must have (6:260). His aircraft application had a flight executive as its application executive, and an engine executive as the aircraft's engine subsystem executive. His flight executive provided these services to the application:

- Obtained outside state information for the aircraft
- Placed aircraft state information on external objects
- Monitored internal aircraft data control
- Managed the interface to the subordinate engine subsystem executive

A.4.2 Concurrency and Temporal Programming. In some application domains, it is possible that some subsystems may execute concurrently. Because of this, the application executive domain model contains structures which control the way in which subsystems execute in a concurrent mode. The application executive prevents causality errors from occurring during concurrent execution by protecting changes to the application clock when the application is running concurrently. Concurrently executing subsystems complicate

the executive's ability to model time during each application's execution. If processing time is an attribute of a subsystem and the global clock is updated with the subsystem processing time after a subsystem executes, the executive must figure out how to update the global clock after concurrent process execution (29:141). Clearly, the application executive domain model should provide a means to consistently update the application clock. Also, in order to model time in the application, each application component requires access to the current time. The application executive domain model must also function in applications where time is not a factor.

A.4.3 Domain Specific Application Executive Features. Ideally, this research should define a general application executive which can control any application. In a general application executive, the line between application control and simulation control would always be well-defined. However, it may not be possible to create a general application executive. The application executive must be able to incorporate domain-specific application control information from the application composed by the application specialist. For example, the application executive may use events to signal when I/O must occur or a subsystem must fire. Potentially, the application executive would need to query the composed application to see what application-specific events must occur and how the executive should handle them. Also, the application specialist may want to specify the particular output devices used by the application. This application-specific information must be provided before execution begins. The application executive may be created as a result of examining the composed application and deriving attributes based on the application's attributes; or, the application specialist may answer queries where he or she specifies application information. The application specialist may do both.

A.4.4 Domain Model Services. Synthesis of the information obtained during domain analysis revealed that the Architect Application executive must perform these services:

1. *Event Handling* - The executive services events raised by the application during execution. The executive orders events and may generate events for the application executive and model components.

2. *Registration* - The executive gets application-specific requests for I/O devices and executive services. This involves building executive data structures with information about application subsystems to enable the executive to keep track of each subsystem's execution state. For example, the executive subsystem collects the connections in the composed model and controls them throughout application execution.

3. *Activation* - The executive activates particular subsystems when it's time for them to execute. The order of activation is determined by the schedule and by the current execution state of the component that must execute. Activation is the result of an event and may cause other events to be generated.

4. *Communication* - The executive supervises internal application data transfer. It manages connections between components and provides inputs for processes when it is time for them to execute.

5. *I/O Handling* - The executive supervises application calls for external I/O services. It controls external device interaction with the application. It manages application access to device drivers and buffers for those devices.

6. *Scheduling* - The executive makes an execution schedule for the application. It orders events and determines how to serialize concurrent events.

A.4.5 Executive Domain Model Services vs. J-MASS Executive Services. Chapter 2 contained a section that listed the services provided by the Simulation Run-Time Agent (SRA) in J-MASS. Architect provides similar services. Table 2 lists the services and the application executive domain model primitives which provide them. This table is included to demonstrate that the architect application executive is conceptually similar to the J-MASS SRA.

The application executive domain model differs from the J-MASS SRA in two ways. First, J-MASS contains a special component that keeps track of the location of players

J-MASS Artifact	Domain Model Primitive	Service
Simulation Controller	Event Manager	Event-Handling
Scenario Manager	Registrar	Registration
Synchronizer	Event Manager	Scheduling
Spatial System Manager	Component	—
Interconnect	Connection	Communication
Locus of Control	Component	—
Activator	Object Control Block	Control Flow
Journalizer	Device	I/O Handling

Table 2. J-MASS Services Compared to Application Executive Domain Model Services

in the model called a Spatial System Manager. If an application specialist wants to implement a model in Architect where the location of the components mattered, he would create his own subsystem to keep track of this information. Secondly, J-MASS uses a special structure called a *Journalizer* to obtain simulation data. The Architect application executive domain model requires the application specialist to insert a device in his application during composition if he wants to send data about the execution to devices external to the Architect run-time environment.

A.5 Choose Model Representation

There are many benefits to formal system specification (10:19). Formal specification eliminates requirements ambiguity and forces system designers to focus on what a system should do prior to implementing the system. The goal of this domain analysis process was to produce a domain model of the domain of application executives that described the entities in an application executive and the ways that these entities may change during execution. The choice of a domain model representation technique depends on the characteristics of the scoped domain of application executive for Architect, REFINE implementation constraints, and available formal specification techniques.

A.5.1 Formal Specification Techniques. Potter, Sinclair, and Till break down the many possible techniques of formal specification into three basic families (4:273–274). State based specification languages such as Z, REFINE and the Vienna Development Method

(VDM) use set theory and predicates to construct system models. Process algebras like Communicating Sequential Processes (CSP) model a system as a collection of processes which communicate. According to Potter, Sinclair and Till, CSP is useful when modeling systems where interprocess communication is emphasized (4:273). Models constructed from algebraic techniques consist of a series of equations which express model components and behavior. Structured analysis methods rely on a graphical representation to express system components and behavior.

A.5.2 Scoped Domain Characteristics. Section A.3 outlined the specific constraints placed upon Architect's application executive by Architect's implementation and research goals. Section A.4 discussed the key characteristics of the scoped domain and stated that these would be the main clues as to what objects would be part of an application executive model. The characteristics of the scoped domain also influence the domain modeling language in this domain analysis process. When searching for a domain modeling language, one must decide which features of the domain of interest should be emphasized. Prieto-Díaz refers to this as defining the goals of domain analysis (16:67). In the case of the domain of Architect application executives, the domain model representation will concentrate on keeping track of the execution state of the application. This is because the Architect implementation of the OCU model structures Architect applications into sets of subsystems that maintain the state of their subordinate primitive objects. The application executive could be structured into a group of related subsystems as in (3). However, that type of encapsulation of services into subsystems supposes that there are primitive objects defined which provide these services that can be grouped into subsystems. A state based specification language with set-theoretic types, like REFINE or Z would be suitable as an application executive domain modeling language.

A.5.3 REFINE *Implementation Constraints.* The application executive domain model was formalized and converted into executable code and incorporated into Architect. Thus, a key reason for choosing a particular domain modeling technique was the ease of its transformation into REFINE, Architect's implementation language. As stated in Section A.5.2, REFINE is a specification language itself. REFINE allows a programmer to

specify entities in the domain as objects with attributes. REFINE contains first order predicate calculus constructs. These constructs allow a programmer to specify pre-conditions and post-conditions which can be satisfied automatically using a REFINE transform construct. REFINE also contains a means to organize objects into tree structures and traverse them easily. Clearly, an appropriate domain model representation should be simple to transform into REFINE and incorporate into Architect.

A.5.4 Domain Model Representation. In this domain analysis process, the domain model representation includes the choice of a formal specification language and a description of the artifacts which will be defined using this formal specification language. The domain analyst uses the specification language and the artifacts to capture the objects and operations in the domain model.

A.5.4.1 Specification Language Choice. In Architect, the application executive must manage the execution of subsystems. These subsystems have states and the application executive must control the way these states change. Therefore, a state based specification language would be the most suitable kind of domain model representation language. REFINE is the most suitable specification language available for this domain analysis process for three reasons. First, REFINE is already Architect's implementation language. Using REFINE eliminates the translation step between the specified and implemented domain model. Second, REFINE can be compiled. While the compilation process does not ensure the model's correctness, it serves as an automated type consistency check for the domain model. Finally, REFINE contains an automated tool called DIALECT for defining and parsing domain specific languages. The domain model can be defined in terms of this domain specific language. Architect utilities can parse this defined model into the REFINE technology base thereby creating instantiated objects that can be manipulated by Architect system software.

A.5.4.2 Domain Artifacts. The application executive domain model was built in stages. Model development began by listing Architect application executive services. Rumbaugh object models described the objects and operations which carry out

these services. When necessary, Rumbaugh dynamic and functional models provided additional information as to how these service's objects change state or process information. (A complete description of the Rumbaugh object-oriented analysis technique may be found in (11).) The Rumbaugh object models became encoded in REFINE object declarations. The operations listed on the object model were encoded as REFINE functions. A domain-specific language was defined which described these REFINE objects and attributes.

A.6 Identify Objects

Before identifying objects, it was necessary to define what the addition of an application executive adds to the Architect system. The application executive controls the flow of data and the flow of control through the application. It relies on the logic contained in each model component to manage data and control flow below the component level. An application consists of an application executive together with a model composed from a particular domain. The application specialist must specify how the application must behave and what subsets of application state will be recorded as application output. One must consider what objects help ensure correct component state changes when identifying objects which make up an application executive domain model. The application specialist must choose the correct model components to answer the following questions:

- The level and method of subsystem interaction — do two closely related subsystems, say a car and a trailer, need to communicate with another component in order to drive down a road?
- The desired application output technique — how will the application specialist view application behavior? Will he or she output data to a file, a common window, or the Refine interaction window?
- The required method of application input — how will the application specialist give external input to the application?

The application executive provides services for four different types of applications. These types of applications differ in the way in which they execute. These executive modes are:

- Event-driven sequential
- Event-driven concurrent
- Time-driven sequential
- Time-driven concurrent

An event-driven model executes as the result of servicing events which are asynchronously raised by the model components and the executive. A time-driven model is a model in which each component reacts to a change in the clock. A sequential model is a model where only one application component can execute at a time t. A concurrent model permits multiple subsystem execution at any one time. The application executive will provide its services in different modes by utilizing different executive model components to handle these different modes of operation.

A domain model of an Architect application executive consists of the following objects and attributes. Note that each object's attributes are listed below it.

Activate Event - A kind of internal event which notifies the object control block (OCB) it may change its execution state to either RUNNING or BLOCKED. The event manager determines which state the *activate* event will change the OCB to based on the current OCB state. RUNNING signifies that a component is to begin execution. BLOCKED signifies that the component is waiting for valid input data, or an external device to be able to take its output.

- *OCB-Name : Symbol*

Application Event - This possibly abstract class describes a kind of internal event raised by an application that must be serviced in some application-specific or architecture-specific way. For example, an airplane simulator may raise this event when the aircraft crashes. An OCU-type model component may raise an *update* or *set-state* event. The mapping between this event and its service routine is established by the component that raised it.

Application Executive - The top-level, abstract object that is composed of these other objects.

- *Mode: Mode-Types = (Event-Driven-Sequential, Event-Driven-Concurrent, Time-Driven-Sequential, Time-Driven-Concurrent, Non-Event-Driven-Sequential)*

Clock - This abstract class maintains application time. Advanced by the event manager to time $t + \tau$ following the service of all possible events at time t.

- *Time : Integer*

Component - An application model component. The component's execution results from the service of an *application* event. A component's execution state is maintained by an object control block.

- *Name : Symbol*

Connection - A link between components (or between a component and a device) which buffers the data which flows between them. The connection object raises a *receive* event when it gets new data. This notifies its sink object that it had new data which must be considered by the sink object as it executes. A connection cannot exist without a source object and a sink object. *Export-Obj* and *Import-Obj* refer to OCU-specific sources and sinks for data in an Architect application.

- *Source : Export-Obj*
- *Sink : Import-Obj*
- *Status: Connection_Status_Type = (CONSUMED, NOT-CONSUMED)*
- *Old-Data : Any-Type*
- *New-Data : Any-Type*

Device - This object class represents an external device. This object's methods are drivers for an external device. Note that specializations of this object class may include file, X-Window, or common window objects. Thus object writes data to the application and reads data from the application. Conceptually, this object is similar to a component. Its operation state is maintained by a device control block object.

- *Name : Symbol*
- *Info : Buffer*

Device Control Block - The device control block (DCB) object controls a external device and acts as the interface between the application and the external device. This object responds to *transmit* events raised by its associated device when device wishes to write data to its connections. The DCB responds to *receive* events raised by the associated device's input connections.

- *Device-Name : Symbol*
- *Input-Stat : Connection_Status_Type*
- *Output-Stat : Connection_Status_Type*
- *Device-State : Device_State_Types = (BLOCKED, IDLE, INPUT-PRESENT, OUTPUT-PRESENT)*

Done Event - A type of executive event which signifies that a component will not raise any more events at time t, and it is safe to increment the executive clock.

Ed-Clock - This concrete class maintains application time in event-driven applications. It does not raise *transmit* events to provide time service to other components in the application.

Ed-Seq Event Manager - This concrete class services events raised during event-driven sequential execution. It differs from the time-driven event-manager in that it accepts all types of events from application components.

Event - This object class describes when application state has changed. This event requires servicing by either changing the execution state of an object control block, internal transfer of data, external transfer of data, beginning execution, or ending execution. *Object-Name* and *Subsystem-Names* are used by Architect to determine which

- *Object-Name : Symbol*
- *Subsystem-Names : seq(Symbol)*
- *Ev-Time : Integer*
- *Priority : Integer*

Event Manager - This abstract class services events raised during application execution in the order dictated by the schedule. It also raises the *executive* events necessary to service *application* events and to ask for executive functions. The *Old-Event-List* contains a list of events, as serviced by the event manager, in their order of service.

- *Current-Event : First(Schedule)*
- *Event-List : Schedule*
- *Old-Event-List : seq(Event)*

Executive Event - This abstract class describes a kind of event which requests an executive service such as moving data or control through an application.

New-Data Event - This event, a specialization of the *executive* event notifies an application component that it should check its input area for new data.

Object Control Block - This object keeps the execution state for the component. When a component's execution state is RUNNING, it may generate events, accept input, and produce output. When a component's execution state is BLOCKED, it is waiting on valid input or it is waiting to write its output to its connection which is itself waiting on an external device. An INACTIVE component is done executing for good, or it has not been cued to execute yet by an activation event. The event manager uses the data contained in the OCB to determine how it should handle executive events. For example, the event handler behaves differently during an activation event when the Input_Valid field is false than when it's true.

- *Component-Name : Symbol*
- *Input-Status : Boolean*
- *Output-Status : Connection_Status_Type*
- *Execution-State : Execution_Types = (BLOCKED, RUNNING)*

Receive Event - A kind of internal event which tells an object control block that new input its component needs has arrived in its input connection. This event is raised by a connection. The *Receive-Import-Name* attribute denotes the data's target *Import-Obj*.

- *Receive-Import-Name: Symbol*

Registrar - This object contains a mapping between components in the application and the services they have requested. It uses this mapping to create objects necessary to perform those executive services for subordinate subsystems when necessary.

- *Where-Services : set(components, exec-obj)*

Remove Event - This object is a subclass of *executive* event. It notifies the event manager it must remove events of a particular type which pertain to a specific component from the schedule. If the target event has any special attributes associated with it, then *Remove-Val* contains those.

- *Remove-Val : Name-Value-Obj*
- *Event-Type : Symbol*

Schedule - An abstract object consisting of the total ordering of events in the application. The schedule is created and modified by the event handler.

Start-Event - A kind of *executive* event which signifies that the event manager should begin operation. This event also sets the clock to the start time indicated in the event.

- *Start-Time : Integer*

Stop-Event - A kind of *executive* event which signifies that the event manager must halt execution at the time indicated in the stop time field.

- *Stop-Time : Integer*

Td-Clock - This concrete class maintains application time in event-driven applications. It raises *transmit* events to provide time service to other components in the application.

Td-Seq Event Manager - This concrete class services events raised during time-driven sequential execution. It differs from the event-driven event-manager in that it accepts only *executive* events from application components.

Transmit-Event - A kind of *executive* event which signifies that an application component needs to send information to a connection. The information in the connection object allows the connection to get the information from its source component. However, the connection manager needs the information contained in the *transmit* event to find the proper connection.

- *Transmit-Export-Name : Symbol*
- *Transmit-Subsystem-Name : Symbol*
- *Transmit-Primitive-Name : Symbol*

Figure 13 represents the objects identified in this section. This graphical representation was transformed into REFINE object declarations during the *Abstract Objects* steps of this domain analysis which are documented in Section A.8.

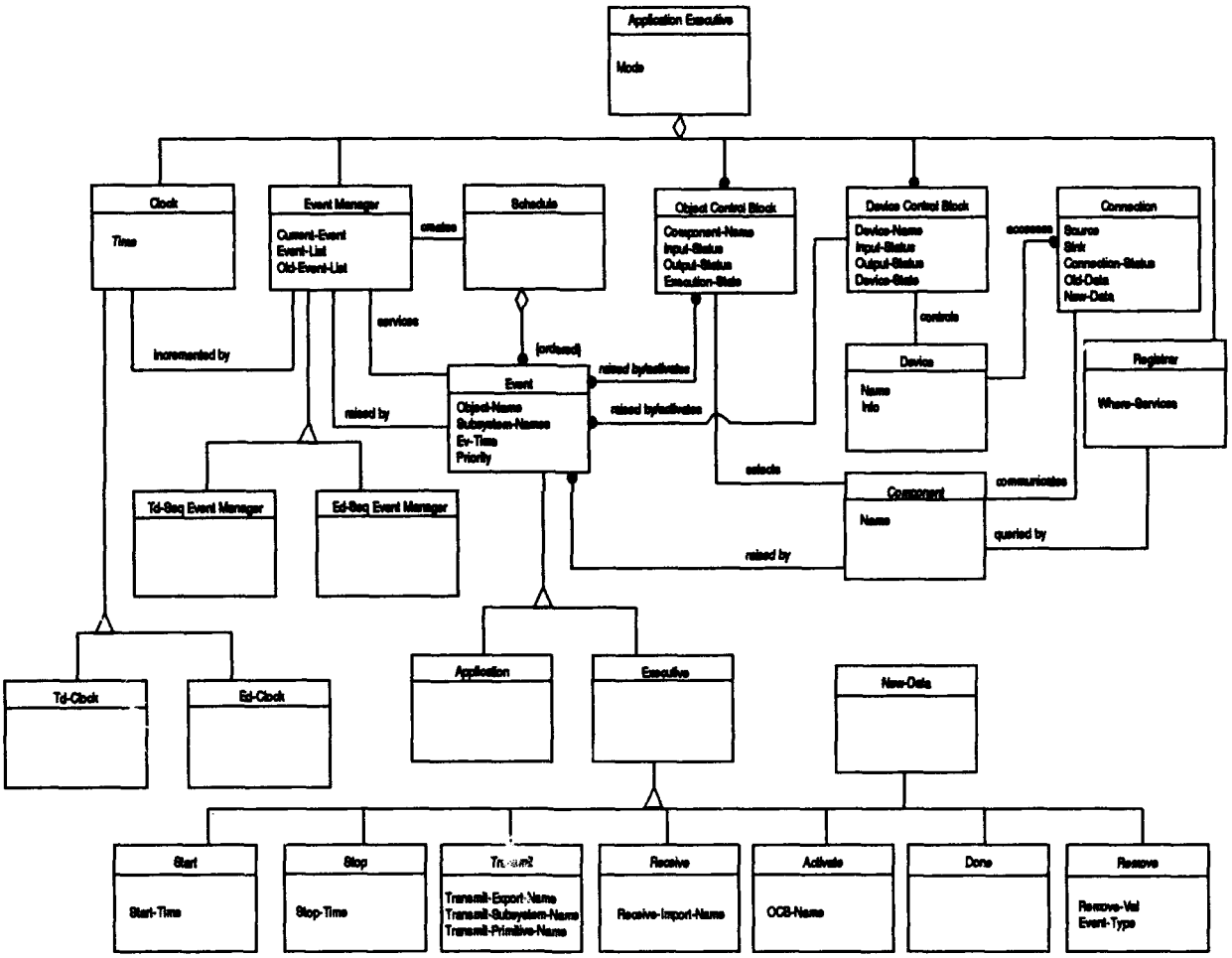

Figure 13. Application Executive Object Model

A.7 Identify Operations

This section names the operations associated with each object class. Operations are object methods which can be performed on each object and change that object's state. These operations result from using event scenarios to describe object interaction. These scenarios are represented graphically as dynamic and functional models which appear in this section.

A.7.1 Scenarios. These scenarios describe the operation of the application executive domain model. They were written to validate the structure of the application executive object model and to discover the dynamic relationships between the objects in the domain model. They do not literally describe the operation of the formalized model components in Architect. They contain notional descriptions of how a concurrent application's executive might work, if the concurrent domain model primitives were formalized and included into Architect. These scenarios were used to create the functions implemented by the formal domain model primitives, and they document design decisions made while getting ready to formalize the domain model.

These scenarios describe the two application executive phases: registration and execution. During the registration phase, the application executive determines which components need executive services during model execution. The executive creates objects which provide these services for each component that requests them. Then, the application executive enters the execution phase when the event manager services various types of events beginning with the *start* event. In event-driven mode, the event manager orders the events it services based on time and by priority for those events which are scheduled for the same time. Why have priority events? Since a component may want to update itself based on the state of a subordinate component at that time, the subordinate component *application* events must be serviced before the higher level events that are tagged with the same time. Also, an application specialist may want to use priority events to handle error or other special conditions which could arise during execution. In time-driven mode, each component is notified of a clock change and it reacts to the correct time value by either executing or not executing at that time. The event manager changes the clock periodically. The appli-

cation specialist specifies when the clock changes by scheduling regular clock *application* events. For example, the application specialist may schedule a *start* event for time zero. Then, he may schedule clock *application* events every second thereafter. After each clock change, the application specialist schedules *application* events for each component, which contain the current time. In sequential mode, these events are serviced in sequence. In concurrent mode, these events are serviced concurrently. Note that in time-driven mode, the components do not schedule *application* events for themselves. Thus, the application operates in a synchronous fashion. In both modes, the executive's execution phase concludes when the event manager services the *stop* event. (Note that the *stop* event can be raised by the user as well as by the application.)

The four different modes of executive operation affect the ways in which the different executive services behave with respect to each other. The registration and execution scenarios listed below represent how the executive domain object model behaves in all modes. Explanations of mode dependent behavior changes in the event manager, the components, and the event service routines are included in these scenarios. The scenarios listed below which apply to the execution phase are broken into several cases for simplicity. These additional scenarios describe what the executive event manger does when it services each type of executive event in each of the four modes.

A.7.1.1 Registration Scenario in All Modes. The following events occur during the registration phase:

1. User starts registrar.
2. Registrar queries user for desired mode of operation.
3. Registrar queries components for executive services required by the components.
4. Components return required services.
5. Registrar creates objects to provide those services, considering the mode the user selected.
6. Registrar loads event manager with *application* events that will activate components at start time.
7. Event manager shows the user these events.
8. User may change this by adding/deleting events from the schedule.

9. User starts event manager by specifying a start time and externally raising a *start* event.
10. Event manager services the *start* event by setting the global clock to the start time.
11. Event manager deletes the *start* event, the highest priority event at the start time, from the schedule.
12. Event manager services all events with a time stamp equal to the current time on the clock in priority order (note that now the application executive enters the execution phase).

A.7.1.2 Execution Scenario in Event-Driven Sequential and Event-Driven Concurrent Modes. The following steps describe the application execution phase in all event-driven modes. During application execution, the event manager is operating and is servicing all events that components and devices raise in the application. Components and devices are only activated at a particular time if they or the application specialist schedule *application* events for them. Activation occurs when the component's object control block is told to change its state to either running or blocked, depending on the condition of the component's inputs. The feed-forward nature of data transfer between components drives component activation when components get input from a connection (see Sections A.7.1.10 and A.7.1.8 on *transmit* and *receive* event services). The scenario assumes there are model components which are concurrently executing and generating events at the component and subcomponent levels. Events in the schedule (event list) are ordered by time and then by priority. The component's place in the component/subcomponent hierarchy normally determines the priority of its events, although the application specialist may designate high priority components whose events must be serviced first. The *executive* events *stop* and *start* always have the highest priority.

1. Event manager services all events at the absolute simulation time t in order of priority.
2. During execution, the components and their subordinate components and subcomponents raise events. These events are scheduled for relative times $\tau_0 \ldots \tau_j$, (where j is the total number of events raised) and passed up to the event manager.
3. Event manager stamps each event with the correct absolute global time $t + \tau_i$ (where $0 \leq i \leq j$) and inserts them in the schedule.
4. In concurrent mode, all top-level components signal that they will send no more events during simulation time t by sending a *done* event up to its parent component stamped with a relative time of zero.

5. In concurrent mode, event manager services the *done* events by changing the applicable component's execution state to INACTIVE.
6. In concurrent mode, when all top-level components are INACTIVE at time t, and there are no more time t events on the schedule, it is now safe to increment the clock. The event manager sets the clock to the time indicated by the event at the top of the schedule. This event is the next one that is to occur in absolute time.
7. In sequential mode, the components do not need to check in when they are done producing events at time t.
8. In sequential mode, the event manager looks on the event list to see if all time t events have been serviced. If they have been serviced, then it is safe to increment the clock in the same manner as in the concurrent mode.
9. Event manager services the events in priority order with a time equal to the time shown on the clock.
10. Event manager continues servicing events, counting *done* events, and incrementing the clock in this manner until it receives a *stop* event.

The scenario described above highlights the differences between the event-driven concurrent mode and the event-driven sequential mode. In both modes, components begin executing when an *application* event which pertains to the component is serviced and their associated object control blocks have the correct execution state. Object control blocks that are in the RUNNING state at the same time indicate that their components are executing concurrently. Concurrency is simulated by not allowing simulation time to advance until all time t events in the application have been serviced. In concurrent mode, the event manager knows when all time t events have been serviced when it receives the proper number of time t *done* events (thus there are no more RUNNING components) and there are no more t events on the schedule. This is because a component cannot raise events unless it is RUNNING. If a component has nothing to do at that time it raises a *done* event. This scheme implies that each component is able to determine when it is done generating events "for now." Sequential operation only requires that all time t events raised by each component are serviced before the event manager can increment the clock. Counting *done* events is not necessary because there is only one thread of control.

The scenario mentioned priority and that events will be ordered by both time and priority. The event manager needs to order events by priority because low level components may need to complete their time t state change prior to upper level components if those upper level components depend on the lower level components state at time t to compute

their state at time t. Priority-ordered events allow the application specialist to specify failure events raised by a low-level subcomponent that must be handled immediately at the highest level in the application. Also, the other executive services will raise events that will be serviced by the event manager. These executive events need to be serviced before *application* events scheduled for the same time.

A.7.1.3 Execution Scenario in Time-Driven Sequential and Time-Driven Concurrent Modes. In the time-driven mode, the application executive must give each component a chance to respond to changes in the clock time. In other words, the clock makes things happen in the application, and "sequential" and "concurrent" describe two different ways for the clock to make things happen. Here is how the application executes in a time-driven concurrent fashion:

1. In concurrent mode, the event manager notifies each component that the clock has changed with an *application* event containing the current time.
2. Each component determines whether or not it must execute at this clock time.
3. If it must execute, it sends an *activate* event with a delay of zero to the event manager and the event manager changes that component's execution state appropriately.
4. When the component is done executing, it sends a *done* event to the event manager to change its execution state back to INACTIVE.
5. When all components have signalled that they have had a chance to respond to the clock, the event manager changes the clock to the next increment.

If the application is executing in a sequential manner, the application executive must place component execution in some order. The event manager does this by activating components in a fixed order every time the clock changes. This way, each component does not have to respond back to the event manager with a *done* indication. The event manager reschedules *application* events for valid future times following service of each *application* event. The *event manager* does not reschedule *executive* events. A valid clock time is a time where the event manager can place an event such that no two application components are operating at the same time. Round-robin component activation occurs until the event manager reaches a stop time or stop condition.

A.7.1.4 Activate *Event Service.* The application specialist schedules initial *application* events for when each model component should begin execution. *Applica'ion*

event service by the event manager begins by scheduling activation events to allow the executive to determine if the component has an appropriate execution state to begin running. Depending on the state of the component's object control block, activation events may result in blocking a component's execution by blocking its associated object control block because the component's inputs are not ready. Note that these events are only used in concurrent mode. Here is how the event manager services activation events:

1. Check object control block input-valid field to see if it is valid.
2. If it is not then change object control block state to BLOCKED.
3. Else, change object control block state to RUNNING.
4. The event manager re-schedules any *application* events that were saved off because the component was RUNNING or had invalid input when it was originally scheduled to go.

A.7.1.5 Application *Event Service.* During execution, a component or subcomponent may raise *application* events. These events must be serviced by application component methods in the application. They may use application-wide state information to determine how the routines should be serviced. Here is how the event manager services these events in concurrent mode:

1. Event manager uses information in the *application* event to determine which component raised the event.
2. Event manager attempts to change the execution state of the target component by scheduling an activation event for it.
3. The event manager moves the *application* event aside while it services the activation event.
4. With the activation event now serviced, the event manager continues to service the original *application* event.
5. If the activation event indicated that the component was RUNNING before event service, the event manager puts the *application* event aside until it can be serviced.
6. If the activation state is BLOCKED as a result of the activation event, the event manager saves the *application* event for rescheduling when its input becomes valid.
7. If it is RUNNING as a result of the activation event, the event manager sends the event to the target component for the component to finish servicing.
8. Now that the component is RUNNING, it may schedule *transmit* events with results of this execution for some relative times in the future (when the component produces output).

9. It may also schedule *application* events (such as an update event in the OCU's case) for some time in the future, as a result of component execution, as needed.
10. In concurrent mode, schedule a *done* event for a relative time of zero, when the component is done raising events for that time.
11. Event manager services next event.

In sequential mode, service of this event is very simple. The event manager passes control to the component until the component completes execution. There is no need to keep track of the component's execution state.

A.7.1.6 Done *Event Service.* The event manager uses *done* events to establish a safe condition to change the clock by signalling when a component is done executing. Here is how the event manager services a *done* event, an event that is only used in concurrent mode:

1. Change the component's OCB execution state to INACTIVE.
2. Reschedule previously waiting *application* events for the component.

A.7.1.7 New-Data *Event Service.* A *new-data* event is serviced in the same manner as any *application* event. That is, the event manager sends the *new-data* event to the application component that has received new data via a connection. The application component controller uses its logic to determine which subcomponent actually received the new information. The component schedules an *application* event for the correct subcomponent for some time in the future.

A.7.1.8 Receive *Event Service.* In concurrent mode, when an object control block cannot enter the RUNNING state due to having invalid input data, it is BLOCKED until an input connection schedules a *receive* event for the object control block. When the input data becomes valid, the *application* events which appeared when the OCB was blocked are rescheduled and get another chance to execute. Here is how the event manager services a *receive* event in concurrent mode:

1. Set input data to valid for this object control block.
2. If the object control block was RUNNING, set the *receive* event aside until the component is INACTIVE.

3. Else, the connection writes the data from connection to component.
4. Mark the connection CONSUMED.
5. If the object control block was BLOCKED, then reschedule previously blocked *application* event for this object control block at a relative time of zero.
6. Service next event.

In sequential mode, *receive* event service is very simple. The connection manager finds the correct connection object using data in the *receive* event. Then, the connection manager raises a *new-data* event to notify a subsystem that it is to receive new input.

A.7.1.9 Remove *Event Service.* The state of the application may change to the point where some events in the schedule are no longer needed or should not be in the schedule. The *remove* event notifies the event manager that events should be deleted from the schedule. Here is how the event manager services the *remove* event in all modes:

1. Event manager uses information in the *remove* event to delete events from the schedule which are described by the *remove* event.
2. Event manager services next event.

A.7.1.10 Transmit *Event Service.* Components produce output which they place in connections that connect components to components, components to devices, or devices to components. The connection object specifies that each component has one output connection per data item that must go downstream. When a component produces output, it schedules a *transmit* event which contains the output. In the event an upstream component is much faster than a downstream component and that downstream component does not consume the data fast enough, the event manager will ensure the data is not lost. Here is what the event manager does to service a *transmit* event in all modes.

1. Change object control block output to valid.
2. Check connection status.
3. In concurrent mode, if the connection contains data that is NOT_CONSUMED, the event manager will put this *transmit* event aside until it is consumed and exit this service routine.
4. Else, write output from the component to the connection.
5. Mark the connection NOT_CONSUMED

6. Connection schedules *receive* events for the component at the other end of the connection for a relative time of zero.
7. Event manager services the next event in the manner described in the previous sections.

Figures 14 - 21 depict the domain model's dynamic behavior. These diagrams are based upon the textual descriptions listed above.

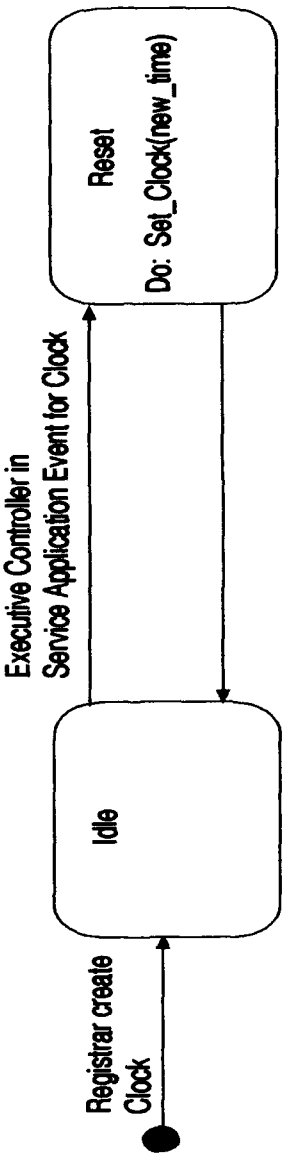

Figure 14. Clock Object Dynamic Model

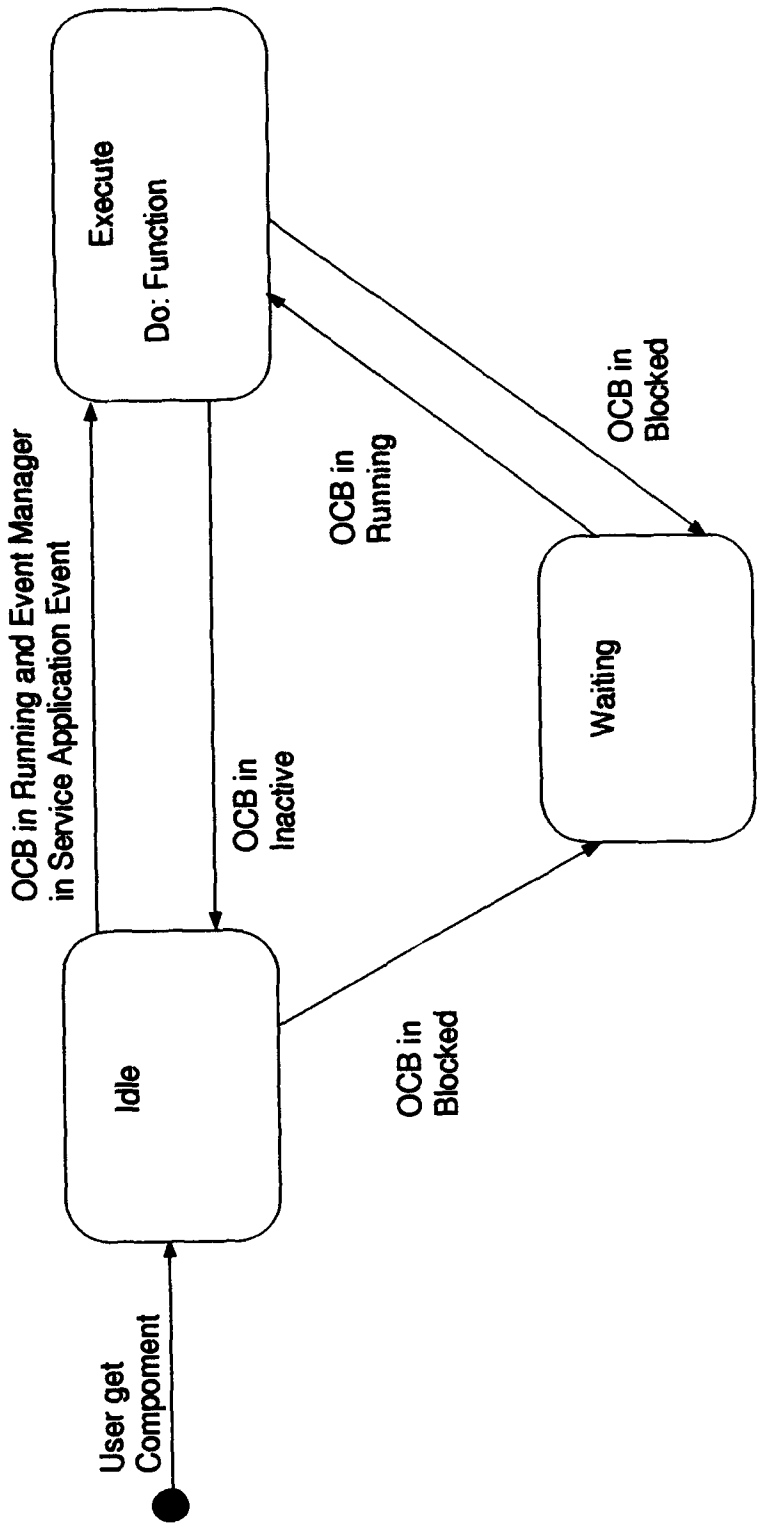

Figure 15. Component Object Dynamic Model

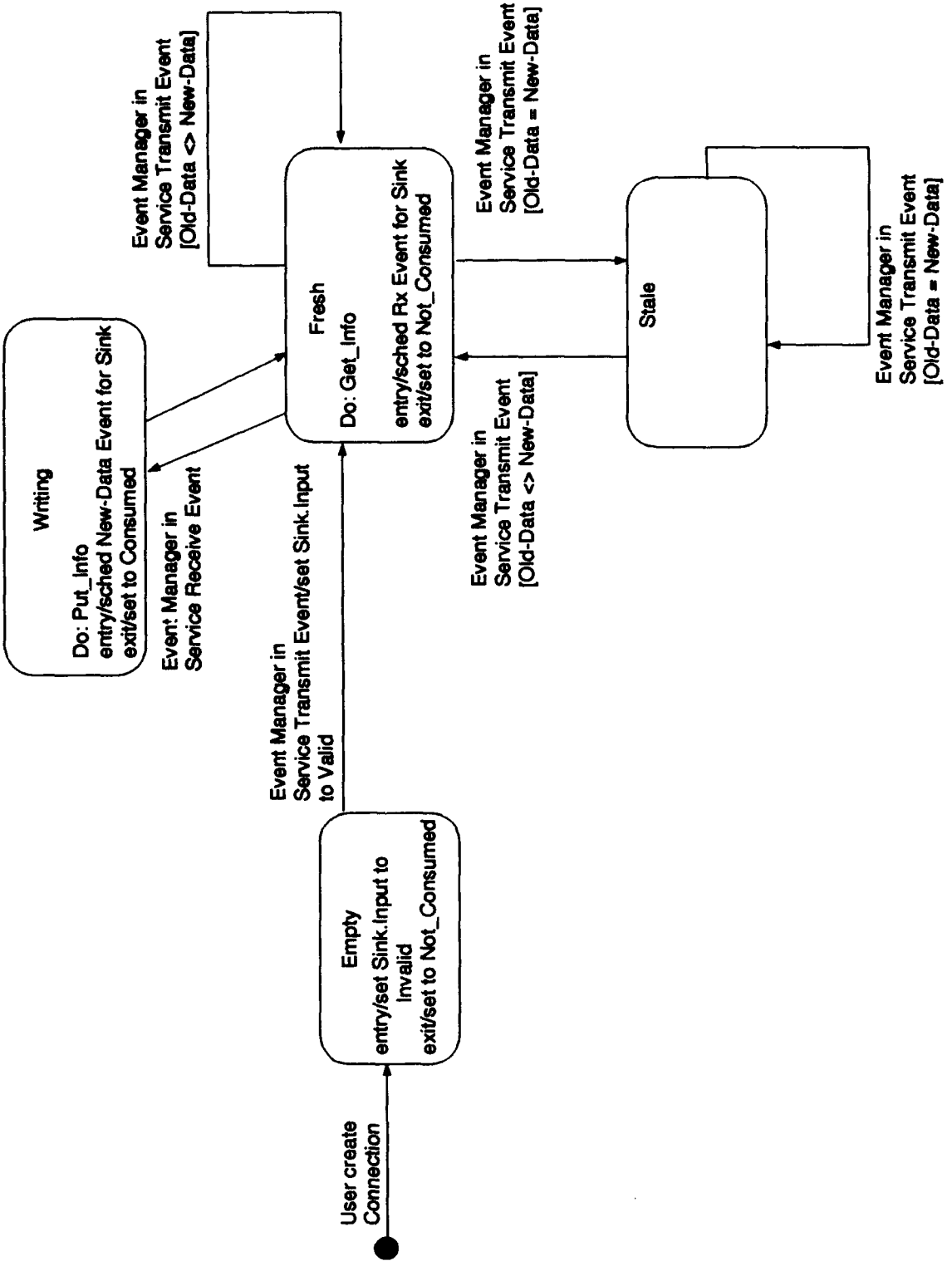

Figure 16. Connection Object Dynamic Model

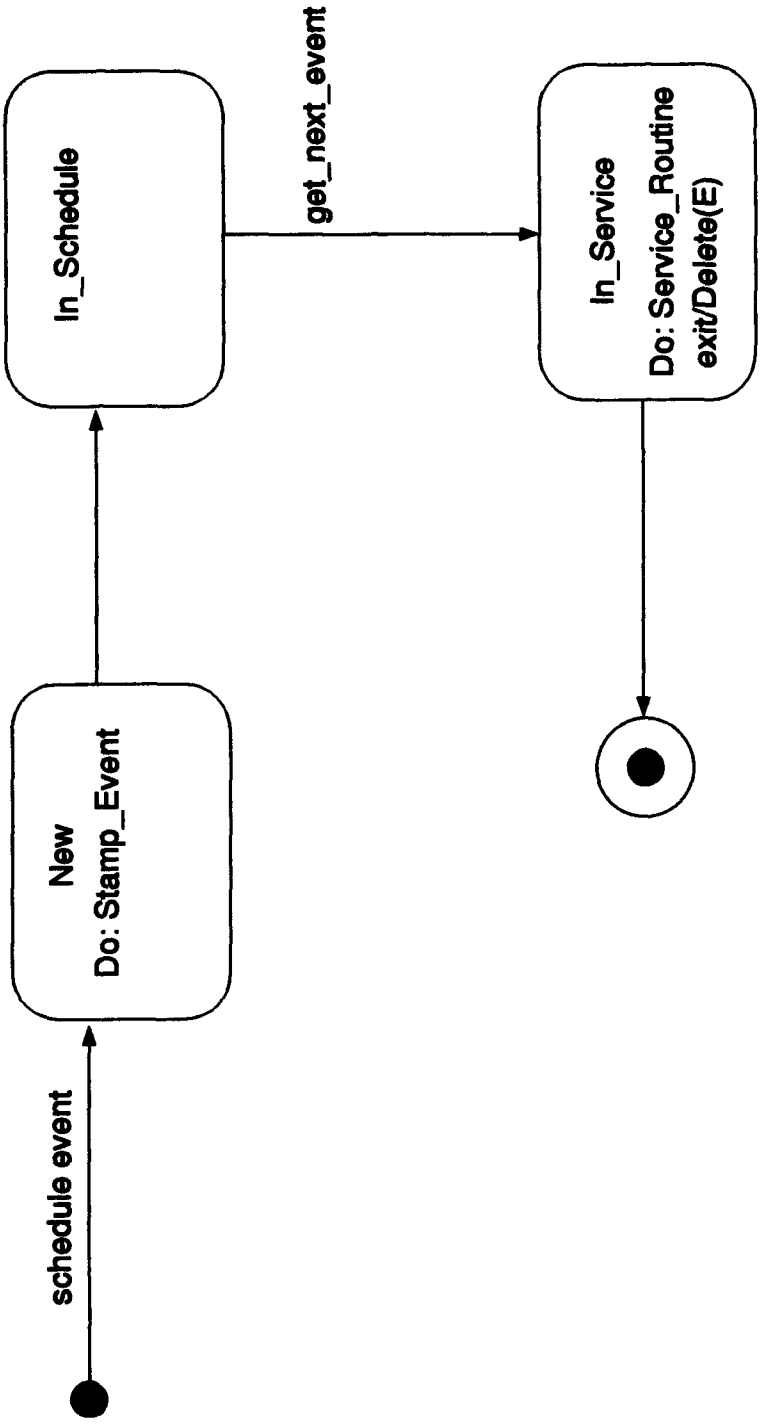

Figure 17. Event Object Dynamic Model

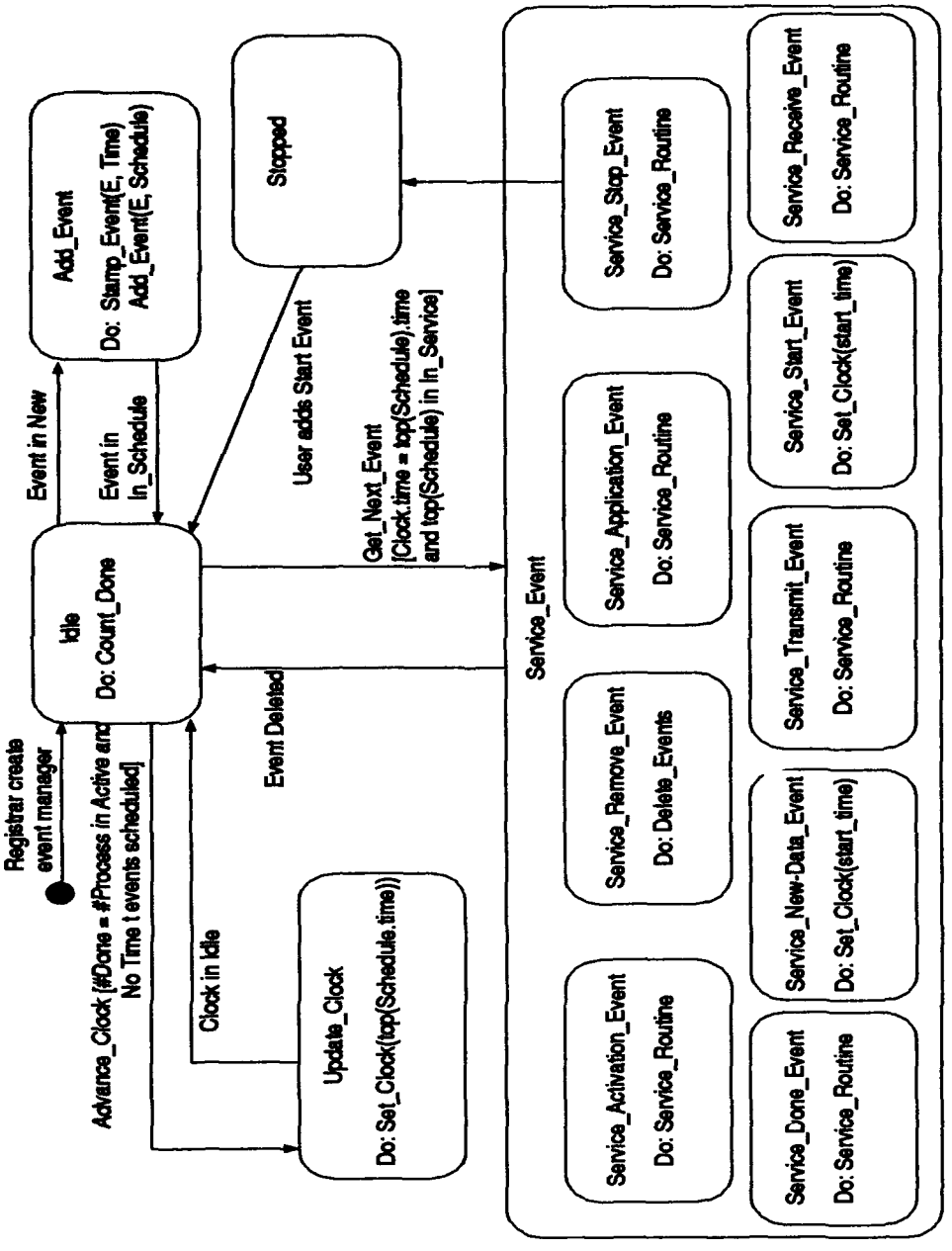

Figure 18. Event Manager Object Dynamic Model

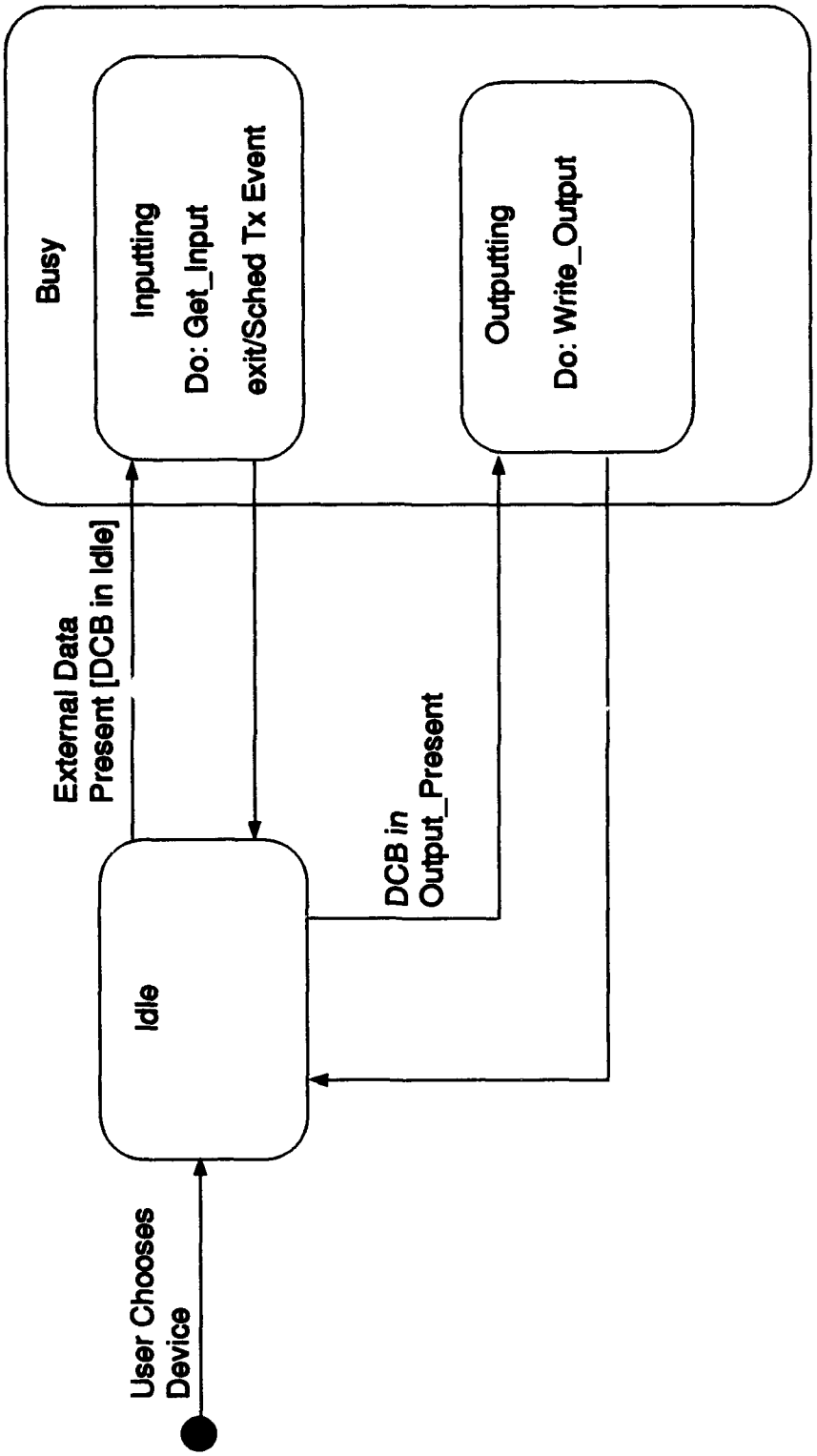

Figure 19. Device Object Dynamic Model

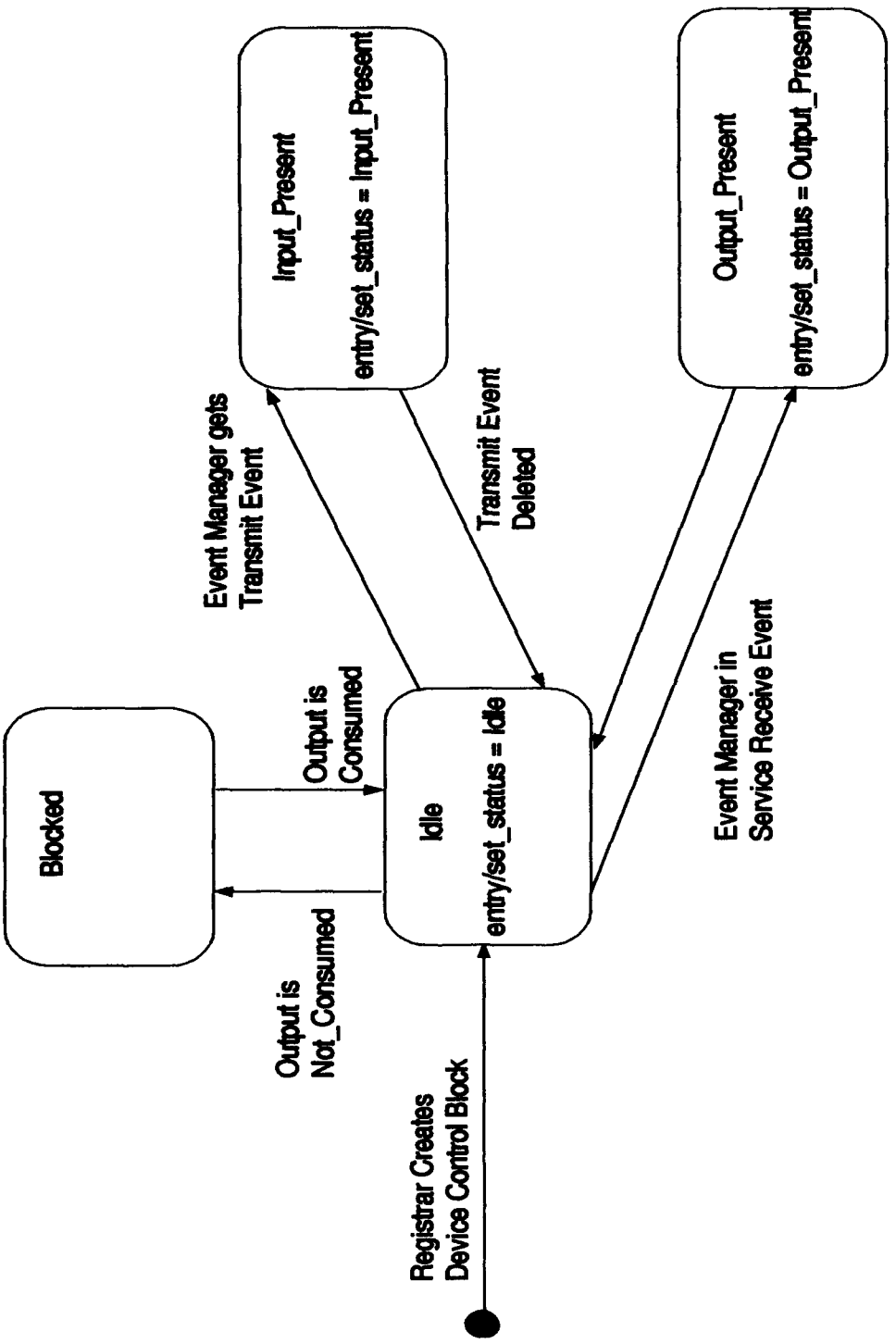

Figure 20. Device Control Block Dynamic Object Model

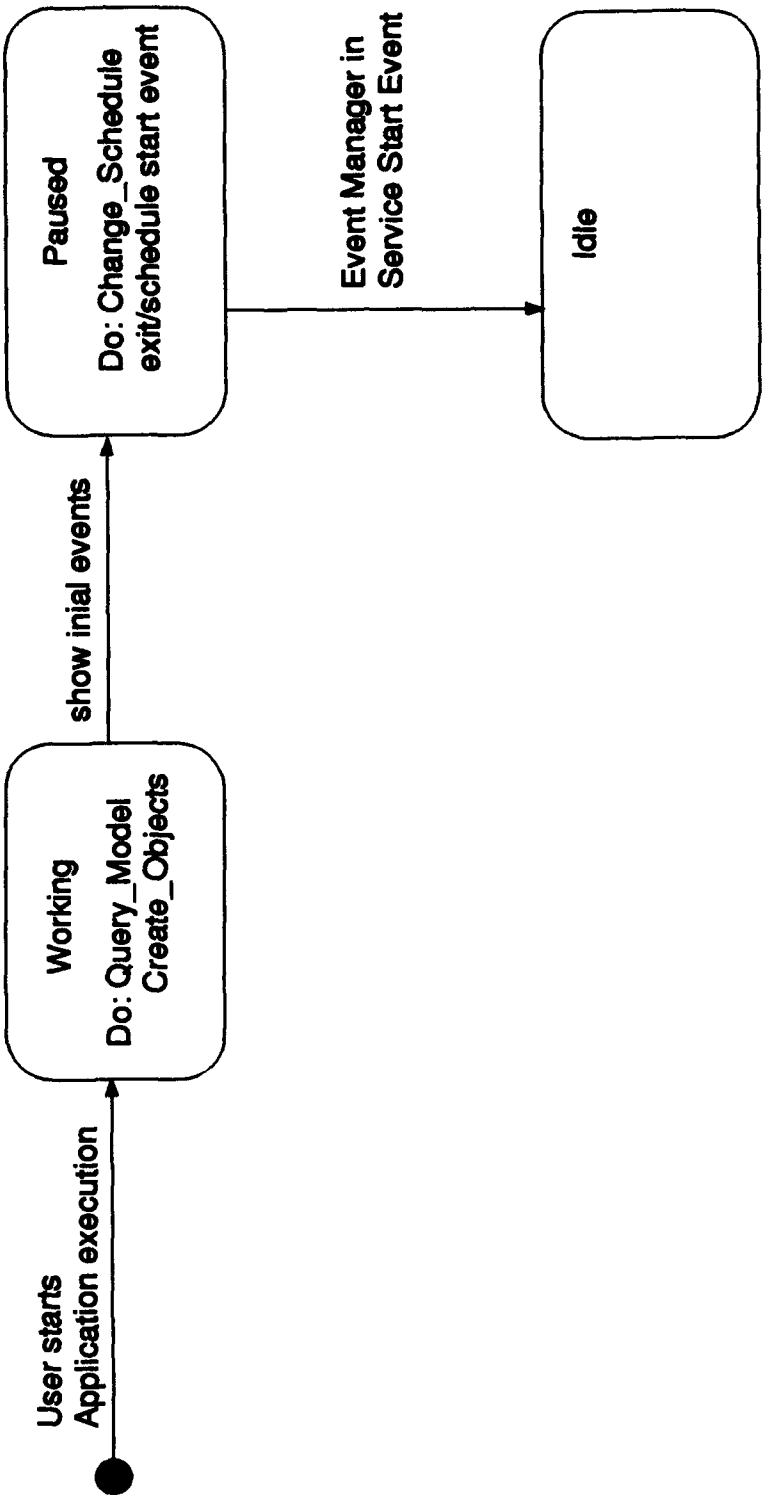

Figure 21. Registrar Object Dynamic Model

A.7.2 Operation Descriptions. This is a description of the operations associated with each object in the application executive object model. Objects that have no unique operations associated with them have been intentionally omitted.

Clock These operations are associated with the *clock* object.

- *Set_Clock* - This operation sets the clock to a specified time.
- *Read_Clock* - This method reads the current time on the clock.

Component The operations associated with a model component are those designated by the domain engineer during component development.

Connection These operations can be associated with the *connection* object.

- *Get_Info* - Get data from its source component and place it in the connection object.
- *Put_Info* - Take data from connection and place it in its sink component.
- *Is_Fresh* - Compare the old buffer values to the new buffer values to see if they are equal. This function returns a boolean value.
- *Set_Status* - Set the connection status to CONSUMED or NOT-CONSUMED.

Device A *device* object carries out these operations.

- *Get_Input* - Get information from an external device.
- *Write_Output* - Give information to an external device.

Device Control Block A *device control block* performs these operations:

- *Set_Status* - Set status of DCB to IDLE, INPUT-PRESENT, or OUTPUT-PRESENT.

Event Manager The *event manager* operates using these functions.

- *Count_Done* - This operation counts the number of *done* events raised by running object control blocks in the application.
- *Advance_Clock* - Advance the simulation clock to the time indicated by the earliest next event once all component object control blocks have checked in with a *done* event and there are no more time t events scheduled.
- *Add_Event* - Add an event to the list of events ordered by time and priority.
- *Delete_Event* - Remove an event from the schedule and place it in the *Old-Events* list.
- *Get_Next_Event* - Get the next event to be serviced.
- *Service_Event* - Call an event's service routine.

Object Control Block The *object control block* performs these tasks.

- *Activate* - Set object control block status attribute to running.

- *Block* - Set a object control block status attribute to blocked.
- *Is_Valid* - A boolean function which indicates if a object control block's input is valid.

Registrar These are the operations performed by the *registrar*.

- *Query_Model* - Query a model for its required executive objects.
- *Create_Objects* - Add the proper number and type of executive objects to the application executive based on the results of *Query_Model*.
- *Change_Schedule* - Allow the user to view and change the application's list of execution events. The user may use this opportunity to add *stop* events to the schedule.

A.8 Abstract Objects

The purpose of this phase in the domain analysis is to identify and express primitives in the domain of application executives. Once the primitives were defined, they were formalized into groups of REFINE object, attribute, and function declarations that followed the format outlined by Randour in (18). This formalization method is outlined in Chapter 5. The results of the formalization are presented in this section by showing the final, implemented object model structure, and by listing each primitive defined during this research effort.

During domain analysis, it became apparent that the event manager object played a central role in application executive operation and that its functional model would be useful during formalization of the domain model. The object model of the application executive contained connections, but contained no coherent way to manage them. If they are to be controlled by the executive, the mission of controlling them should be encapsulated in a primitive. The new primitive, the *connection manager* was also modeled functionally as it was formalized. Tables 3 - 5 depict the functional models used to create the formal primitive objects presented in this section.

Operation	Service Transmit Event
Precondition	$\exists_{x,y}$: (x is Transmit-Event ∧ y is Connection-Obj ∧ y in Connection-List ∧ Transmit-Export-Name(x) = Export-Name(Source-Export(y)) ∧ Transmit-Subsystem-Name(x) = Export-Owner-Sub(Source-Export(y)) ∧ Transmit-Primitive-Name(x) = Producer(Source-Export(y)))
Postcondition	(Connection-State(y) = Not-Consumed ∧ Old-Data(y) = Export-Value(Source-Export(y)) ∧ Receive Event Scheduled for Sink-Import(y))
Operation	Service Receive Event
Precondition	$\exists_{x,y}$: (x is Receive-Event ∧ y is Connection-Obj ∧ Receive-Import-Name(x) = Import-Name(Sink-Import(y)) ∧ Object-Name(x) = Consumer(Sink-Import(y)))
Postcondition	$\exists_{x}$: (x is New-Data-Event Scheduled for Sink-Import(y)) ∧ Connection-Status(y) = Consumed ∧ Import-Changed(Sink-Import(y)) ∧ Import-Value(Sink-Import(y)) = Old-Data(y)

Table 3. Connection Manager Functional Model

Operation	Start
Precondition	$\exists_{x}$: (x is Start-Event ∧ x is First(Schedule))
Postcondition	$\exists_{x}$: (x is Start-Event ∧ x is last(Old-Events) ∧ Current-Time = Start-Time(x))
Operation	Service an Event
Precondition	$\exists_{x}$: (x is First(Event-List) ∧ Ev-Time(x) = Current-Time)
Postcondition	x = Last(Old-Events)
Operation	Add Event
Precondition	$\exists_{x}$: (x is Event) ∧ (first(subsystem-names(x)) ≠ name(executive)) ∧ (x ¬ in event-list ∨ x ¬ in Old-Events)
Postcondition	(x in Event-List)
Operation	Increment Clock
Precondition	$\exists_{x}$: (x is First(Event-List) ∧ Ev-Time(x) ≠ Current-Time)
Postcondition	Current-Time = Ev-Time(First(Event-List))
Operation	Stop
Precondition	$\exists_{x}$: (x is Stop-Event ∧ x is First(Event-List))
Postcondition	Executive Halts

Table 4. Event-Driven Event Manager Functional Model

Operation	Start
Precondition	$\exists_x$: (x is Start-Event $\wedge$ x is First(Schedule))
Postcondition	$\exists_x$: (x is Start-Event $\wedge$ x is last(Old-Events) $\wedge$ Current-Time = Start-Time(x))
Operation	Service an Event
Precondition	$\exists_x$: (x is First(Event-List) $\wedge$ Ev-Time(x) = Current-Time)
Postcondition	x = Last(Old-Events)
Operation	Add Event
Precondition	$\exists_x$: ((x is Event) $\wedge$ (first(subsystem-names(x)) $\neq$ name(executive)) $\wedge$ (x $\neg$ in event-list $\vee$ x $\neg$ in Old-Events))
Postcondition	(x in Event-List $\wedge$ x in Export-Area)
Operation	Increment Clock
Precondition	$\exists_x$: (x is First(Event-List) $\wedge$ Ev-Time(x) $\neq$ Current-Time)
Postcondition	Current-Time = Current-Time + 1
Operation	Stop
Precondition	$\exists_x$: (x is Stop-Event $\wedge$ x is First(Event-List))
Postcondition	Executive Halts

Table 5. Time-Driven Event Manager Functional Model

A.8.1 Application Executive Object Structure.

```
!! in-package("AVSI")
!! in-grammar('user)

var Application-Exec-Obj   :    object-class subtype-of Primitive-Obj

var Connection-Mgr-Obj     :    object-class subtype-of Application-Exec-Obj
var Component-Mgr-Obj      :    object-class subtype-of Application-Exec-Obj
var Device-Mgr-Obj         :    object-class subtype-of Application-Exec-Obj
var Event-Mgr-Obj          :    object-class subtype-of Application-Exec-Obj
var   Ed-Seq-Mgr-Obj       :    object-class subtype-of Event-Mgr-Obj
var   Td-Seq-Mgr-Obj       :    object-class subtype-of Event-Mgr-Obj
var Exec-Clock-Obj         :    object-class subtype-of Application-Exec-Obj
var   Ed-Clock-Obj         :    object-class subtype-of Exec-Clock-Obj
var   Td-Clock-Obj         :    object-class subtype-of Exec-Clock-Obj
```

A.8.2 Executive Domain Model Primitives. This section contains the formal domain models of these primitives:

- Connection Manager
- Event-Driven Clock Manager
- Event-Driven Sequential Event Manager
- Time-Driven Sequential Event Manager
- Time-Driven Clock Manager

A.8.2.1 Connection Manager.

```
!! in-package("AVSI")
!! in-grammar('user)
#||
Filename: connection-manager.re
Author: Bob Welgan
Date: 9 Sep 93

Modifications:

Description:

This file describes a primitive object in the domain of application executives.
The primitive object definition supplied here conforms to the primitve object
template in Anderson's Appendix A.

The connection-manager contains a list of connection objects and the methods
necessary to manipulate the list and correctly change the status and contents
of each connection as indicated by an element of that list.  The conection
```

```
manager raises receive events.

This primitive relies heavily on these declarations which appear in
the OCU domain model
 var CONNECTION-OBJ            : object-class subtype-of World-Obj

 var Source-Exp                : map(CONNECTION-OBJ, export-obj) = {||}
 var Sink-Imp                  : map(CONNECTION-OBJ, import-obj) = {||}
 var Connection-State          : map(CONNECTION-OBJ, symbol)
   computed-using
     Connection-State(x) = 'Empty
 var Old-Data                  : map(CONNECTION-OBJ, any-type) = {||}
 var New-Data                  : map(CONNECTION-OBJ, any-type) = {||}

||#

 var CONNECTION-MGR-OBJ-INPUT-DATA   : set(import-obj) =

       {set-attrs (make-object('import-obj),
                   'import-name, 'in-event,
                   'import-category, 'an-event,    %what kind of event
                   'import-type-data, 'event-obj)} %and which connection

 var CONNECTION-MGR-OBJ-OUTPUT-DATA  : set(export-obj) =
        {set-attrs (make-object('export-obj),
                   'export-name, 'new-event,
                   'export-category, 'an-event,         %events that
                   'export-type-data, 'set-event-obj)}  %connection-mgr produced

 var CONNECTION-MGR-OBJ-COEFFICIENTS : map(CONNECTION-MGR-OBJ, set(name-value-obj))
  computed-using
    CONNECTION-MGR-OBJ-COEFFICIENTS(x) = {}


 var CONNECTION-MGR-OBJ-UPDATE-FUNCTION : map(CONNECTION-MGR-OBJ, symbol)
   computed-using
     CONNECTION-MGR-OBJ-UPDATE-FUNCTION(x) = 'CONNECTION-MGR-OBJ-UPDATE1

% Other Attributes:
 var CONNECTION-MGR-CONNECTION-LIST : map(CONNECTION-MGR-OBJ, set(CONNECTION-OBJ))
   computed-using
    CONNECTION-MGR-CONNECTION-LIST(x) = {}

 form Make-CONNECTION-MGR-Names-Unique
  unique-names-class('CONNECTION-MGR-OBJ, true)


%-----------------------------
%-
%- Service Transmit and Receive
%-
%- This update function responds to transmit, and receive events by moving
%- data from one export area to one import area.
%-
%-----------------------------
```

```
function CONNECTION-MGR-OBJ-UPDATE1 (subsystem : subsystem-obj,
                                     connection-mgr : CONNECTION-MGR-OBJ)
                                                    : set(event-obj) =

 format(debug-on, "CONNECTION-MGR-OBJ-UPDATE on ~s~%", name(connection-mgr));

 let (in-event: event-obj  = get-import('curr-event, subsystem, connection-mgr),
      connection-status : set(symbol) = {},
      event-list        : set(event-obj) = {},
      Found             : Boolean = False)

 format(debug-on, "Connection-manager operating on ~A~%", in-event);

 (enumerate Obj over CONNECTION-MGR-CONNECTION-LIST(connection-mgr) do

%If it's a transmit event that's being handled, then set the CONNECTION
%to the proper values to indicate new data has entered the downstream
%connection(s) and not been consumed.
%conditionally raise a receive event here, if
%the new-data is different than the old data...

     (if Transmit-Event-Obj(in-event) then
      ((((transmit-export-name(in-event) = export-name(Source-Exp(Obj))) &
      (transmit-subsystem-name(in-event) = exp-owner-sub(Source-Exp(Obj))) &
      (transmit-primitive-name(in-event) = producer(Source-Exp(Obj))))
           --> (('Not-Consumed = connection-state(Obj))  &
               (Found) &
               (Old-Data(Obj) <- export-value(Source-Exp(Obj)))));
      (Found -->
          (format(debug-on, "Conn Mgr now servicing Tx event ~A~%", in-event);
           Found <- False;
          (event-list <- event-list with set-attrs(make-object('receive-event-obj),
                            'name, 'RX-2,
                            'subsystem-names, [imp-owner-sub(Sink-Imp(Obj))],
                            'object-name, consumer(Sink-Imp(Obj)),
                            'receive-import-name, import-name(Sink-Imp(Obj)),
                            'priority, 10,
                            'Ev-Time, 0)))))


%If it's a receive event, set the status to consumed, and write
%the value from the buffer to the target import area, raise a new-data
%event, too.  Remeber to set the flag, too

    elseif Receive-Event-Obj(in-event) then
      ((((receive-import-name(in-event) = Import-Name(Sink-Imp(Obj))) &
        (object-name(in-event) = consumer(Sink-Imp(Obj))))
            --> (('Consumed = connection-state(Obj)) &
                (connection-status = {'Consumed}) &
                (import-changed(Sink-Imp(Obj))) &
                (import-value(Sink-Imp(Obj)) <- Old-Data(Obj))));
        format(debug-on, "Conn Mgr now Servicing Rx Event ~A~%", in-event);
        (event-list <- event-list with set-attrs(make-object('new-data-event-obj),
                              'subsystem-names, import-path(Sink-Imp(Obj)),
                              'object-name, first(import-path(Sink-Imp(Obj))),
                              'priority, 10,
```

```
                                'Ev-Time, 0)))));

%determine final connection status

% 'Not-Consumed in connection-status
%                          --> final-con-stat = 'Not-Cunsumed;

   (enumerate Obj over event-list do
    (format(debug-on, "Conn Mgr Raised this event ~A~%", Obj)));

 (if ~Empty(event-list) then
    (event-list <- event-list union
                set-export(subsystem, connection-mgr,
                {set-attrs(make-object('name-value-obj),
                'name-value-name, 'new-event,
                'name-value-value, (event-list intersect event-list))})));

  event-list <- {}; %return no events, not
                    %even the Tx events made
                    %by the set-export command

 event-list



%------------------------------
%-
%- Fill Connection Mgr List
%-
%- This utility adds connection objects to the CONNECTION-MGR-CONNECTION-LIST
%- using the links created by the application specialist during application
%- composition.  It gets these connection objects from the application
%- descriptor object.
%-
%------------------------------

function Fill-Connection-Mgr-List (in-app : spec-obj) =

 format(debug-on, "Filling the Ed-Seq-Connection-List for Application~A~%",
        in-app);


 let(conn-mgr    : connection-mgr-obj = arb({x | (x : connection-mgr-obj) &
                                                 (connection-mgr-obj(x)) &
                                                 (x in kids(in-app))}),
     descriptor  : descriptor-obj = arb({y | (y : descriptor-obj) &
                                             (descriptor-obj(y)) &
                                             (y in kids(in-app))}),
     conn-set    : set(connection-obj) = {})

 conn-set <- Top-Level-Connections(descriptor);
 CONNECTION-MGR-CONNECTION-LIST(conn-mgr) <- conn-set
```

A.8.2.2 Event-Driven Clock Manager.

```
!! in-package("AVSI")
!! in-grammar('user)

#||
Filename:ed-clock.re
Author: Bob Welgan
Date: 6 Sep 93

Modifications:

Description:

This file describes a primitive object in the domain of application executives.
The primitive object definition supplied here conforms to the primitve object
template in Anderson's Appendix A.

The clock contains a time value and the methods necessary to change the time
and output it's value.
||#

% var ED-CLOCK-OBJ : object-class subtype-of Application-Exec-Obj

 var ED-CLOCK-OBJ-INPUT-DATA   : set(import-obj) =
        {set-attrs (make-object('import-obj),
                    'import-name, 'new-time,
                    'import-category, 'time,
                    'import-type-data, 'integer)}

 var ED-CLOCK-OBJ-OUTPUT-DATA  : set(export-obj) =
        {set-attrs (make-object('export-obj),
                    'export-name, 'current-time,
                    'export-category, 'time,
                    'export-type-data, 'integer)}

 var ED-CLOCK-OBJ-COEFFICIENTS : map(ED-CLOCK-OBJ, set(name-value-obj))
  computed-using
    ED-CLOCK-OBJ-COEFFICIENTS(x) = {}

 var ED-CLOCK-OBJ-UPDATE-FUNCTION : map(ED-CLOCK-OBJ, symbol)
   computed-using
     ED-CLOCK-OBJ-UPDATE-FUNCTION(x) = 'ED-CLOCK-OBJ-SET-ED-CLOCK

% Other Attributes:
 var ED-CLOCK-OBJ-TIME : map(ED-CLOCK-OBJ, integer)
   computed-using
    ED-CLOCK-OBJ-TIME(x) = 0

 form Make-ED-CLOCK-Names-Unique
  unique-names-class('ED-CLOCK-OBJ, true)

%------------------------------
%-
%- Function Set-ED-CLOCK
%-
%- This function sets the ED-CLOCK to the time indicated in the import area.
%-
```

```
%-----------------------------

function ED-CLOCK-OBJ-SET-ED-CLOCK (subsystem : subsystem-obj,
                                    the-clock : ED-CLOCK-OBJ)
                                     : set(event-obj) =

 format(debug-on, "ED-CLOCK-OBJ-SET on ~s~%", name(the-clock));

 let (time-value : integer = get-import('new-time, subsystem, the-clock),
      event-list : set(event-obj) = {})

 ED-CLOCK-OBJ-TIME(the-clock) <- time-value;
 format (true, "The current simulation time is  ~d~%", time-value);
 event-list <- set-export(subsystem, the-clock,
                      {set-attrs(make-object('name-value-obj),
                                             'name-value-name, 'current-time,
                                             'name-value-value, time-value)});
 %throw the transmit events away
 event-list <- {};

 event-list
```

A.8.2.3 Event-Driven Sequential Event Manager.

```
!! in-package("AVSI")
!! in-grammar('user)

#||
Filename: ed-seq-event-man.re
Author: Bob Welgan
Date: 6 Sep 93

Modifications:

Description:

This file describes a primitive object in the domain of application executives.
The primitive object definition supplied here conforms to the primitve object
template on page 2, Appendix A, Anderson's thesis.

The schedule contains a list of events and the methods necessary to manipulate
the list.
||#

% var ED-SEQ-MGR-OBJ            : object-class subtype-of Application-Exec-Obj

 var ED-SEQ-MGR-OBJ-INPUT-DATA     : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'simulation-time,
                   'import-category, 'time,
                   'import-type-data, 'integer),

        set-attrs (make-object('import-obj),
                   'import-name, 'new-event,
                   'import-category, 'an-event,
                   'import-type-data, 'set-event-obj)}
```

```
var ED-SEQ-MGR-OBJ-OUTPUT-DATA     : set(export-obj) =
      {set-attrs (make-object('export-obj),
                  'export-name, 'curr-event,
                  'export-category, 'an-event,
                  'export-type-data, 'event-obj),

       set-attrs (make-object('export-obj),
                  'export-name, 'simulation-time,
                  'export-category, 'time,
                  'export-type-data, 'integer)}

 var ED-SEQ-MGR-OBJ-COEFFICIENTS : map(ED-SEQ-MGR-OBJ, set(name-value-obj))
   computed-using
     ED-SEQ-MGR-OBJ-COEFFICIENTS(x) = {}

 var ED-SEQ-MGR-OBJ-UPDATE-FUNCTION : map(ED-SEQ-MGR-OBJ, symbol)
   computed-using
     ED-SEQ-MGR-OBJ-UPDATE-FUNCTION(x) = 'ED-SEQ-MGR-OBJ-UPDATE1

%Ed-Seq-Mgr Object Attributes

 var ED-SEQ-MGR-OBJ-EVENT-LIST : map(ED-SEQ-MGR-OBJ, seq(EVENT-OBJ))
   computed-using
     ED-SEQ-MGR-OBJ-EVENT-LIST(x) = []

 var ED-SEQ-MGR-OBJ-OLD-EVENTS : map(ED-SEQ-MGR-OBJ, seq(EVENT-OBJ))
   computed-using
     ED-SEQ-MGR-OBJ-OLD-EVENTS(x) = []

 var ED-SEQ-MGR-OBJ-CURRENT-EVENT : map(ED-SEQ-MGR-OBJ, EVENT-OBJ)
   computed-using
     ED-SEQ-MGR-OBJ-CURRENT-EVENT(x) = first(ED-SEQ-MGR-OBJ-EVENT-LIST(x))

 form Make-ED-SEQ-MGR-Names-Unique
  unique-names-class('ED-SEQ-MGR-OBJ, true)

%-----------------------------
%-
%- Event-Driven-Sequential-Manager-Object Update Function
%-
%- This rather large primitive update function is broken into three different
%- steps.  The update begins by servicing the current event.  Then, it removes
%- that event from the event list.  Finally, it places new events, raised by
%- the application the executive is controlling, in the event list.  This cycle
%- continues until there are no more events.  These steps are written as
%- separate Refine functions to make them easy to test and read.
%-
%-----------------------------

function ED-SEQ-MGR-OBJ-UPDATE1 (subsystem : subsystem-obj,
                                 event-mgr  : ED-SEQ-MGR-OBJ) : set(event-obj) =

  format (debug-on, " Events are now being serviced by ~s~%", name(event-mgr));

  let (exec-events : set(event-obj) = {},
```

```
      new-events  : set(event-obj) = {},
      Running : Boolean = (~Empty(ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) &
                           ~Stop-Event-Obj(last(ED-SEQ-MGR-OBJ-OLD-EVENTS(event-mgr)))),
      New-Ones : Any-Type = 'undefined,
      Really-New : set(event-obj) = {},
      Sim-Time  : Integer = get-import('simulation-time, subsystem, event-mgr),
      Done : Boolean = False)

(if Running then
    New-Ones <- get-import('new-event, subsystem, event-mgr);
    format(debug-on, "New-Ones are ~A~%", New-Ones);

    (New-Ones ~= 0) -->
      ((if Event-Obj(New-Ones) then
        (Really-New <- Really-New with New-Ones)
        else Really-New <- New-Ones);
      (enumerate Obj over Really-New do
        (if (Event-Obj(Obj) &
             (Obj ~in ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) &
             (Obj ~in ED-SEQ-MGR-OBJ-OLD-EVENTS(event-mgr))) then
      ((new-events <- new-events with Obj);
      (format(debug-on, "Event Manager adding this event from import ~A~%",
              Obj))))));

    % if any new events are valid, add them...
    ~Empty(new-events)
       --> (Ed-Add-Events(subsystem, event-mgr, new-events));

    %service next event
    (Ev-Time(ED-SEQ-MGR-OBJ-CURRENT-EVENT(event-mgr)) = Sim-Time)
          -->  new-events <- Ed-Service-Event(subsystem, event-mgr);

    (enumerate Obj over new-events do
       (Done-Event-Obj(Obj) --> ((Done <- True);
                                 (exec-events <- new-events))));
    ~Done  -->
    (exec-events <- exec-events union {e | (e :event-obj) &
                                    (e in new-events) &
                                    (first(subsystem-names(e)) = 'app-exec)};
    new-events <- setdiff(new-events, exec-events);
%We cannot update the same primitive more than one at any time
    (enumerate Obj over new-events do
     (Update-Event-Obj(Obj) &
      (ex(x) ((x in (ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) or
               x in (ED-SEQ-MGR-OBJ-OLD-EVENTS(event-mgr))) &
           Update-Event-Obj(x) &
           (Ev-Time(Obj) + Sim-Time = Ev-Time(x)) &
           (Object-Name(Obj) = Object-Name(x)) &
           (Subsystem-Names(Obj) = Subsystem-Names(x)))))
                       --> Obj ~in new-events);

%We cannot send the same primitive more than one new data
%notification at any one time
    (enumerate Obj over new-events do
     (New-Data-Event-Obj(Obj) &
      (ex(x) ((x in (ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) or
```

```
            x in (ED-SEQ-MGR-OBJ-OLD-EVENTS(event-mgr))) &
          New-Data-Event-Obj(x) &
          (Ev-Time(Obj) + Sim-Time = Ev-Time(x)) &
          (Object-Name(Obj) = Object-Name(x)) &
          (Subsystem-Names(Obj) = Subsystem-Names(x)))))
                    --> Obj ~in new-events);

    Ed-Add-Events(subsystem, event-mgr, new-events); %add events raised by app

%if service event has not generated a clock update, then check to see if one
%should be generated

    ~ex(x) (x in exec-events &
            Update-Event-Obj(x) &
            (Object-Name(x) = 'global-timer))
          -->  exec-events <- exec-events union
                              Check-Increment-Conditions(subsystem, event-mgr)));

  exec-events

%-----------------------------
%-
%- Ed-Add-Events
%-
%- This mehtod gets all the events raised as a result of application executive
%- subsystem execution and places them in the event-mgr.
%-
%-----------------------------

function Ed-Add-Events (subsystem : subsystem-obj,
                        event-mgr  : ED-SEQ-MGR-OBJ,
                        in-events  : set(event-obj)) =

 let (Done  : Boolean = False,
      index : Integer = 1,
      Sim-Time  : Integer = get-import('simulation-time, subsystem, event-mgr))

  format (debug-on, "New Event Added To Event-Mgr ~A~%",in-events);

  %add the current absolute time to each event's relative time in the
  %set of incoming events

  (enumerate new-event over in-events do
    (Ev-Time(new-event) <- (Ev-Time(new-event) + Sim-Time)));

  %take each event object and place it in the event-mgr one at a time
  %the event must either go at the head of the sequence, in the middle,
  %or at the end.
  %find out where it should go...

 (enumerate new-event over in-events do
   (while ((index <= size(ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))) & ~Done) do

    ((Ev-Time(new-event) < Ev-Time((ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))(index)))
       --> (Done);
```

```
    (Ev-Time(new-event) > Ev-Time((ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))(index)))
      --> (index = (index + 1));

    ((Ev-Time(new-event) = Ev-Time((ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))(index))) &
    (Priority(new-event) <= Priority((ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))(index))))
       --> (index = (index + 1));

    ((Ev-Time(new-event) = Ev-Time((ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))(index))) &
    (Priority(new-event) > Priority((ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))(index))))
       --> (Done))); %end while

  %Place the new event object at the location pointed to by index
    ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)
          <- insert(ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr),index, new-event);
    Done <- False;
    index <- 1)

%------------------------------
%-
%- Check-Increment-Conditions
%-
%- This checks the current event manager imports and state variables to see
%- if it is the right time to increment the clock.  If the conditions indicate
%- the clock should be set to a new time, this function sets an export area to
%- a new time.  Note that service of a start event invovles an un-conditional
%- update of the event manager's time export, and is not considered here.
%-
%------------------------------

function Check-Increment-Conditions (subsystem : subsystem-obj,
                                     event-mgr : ED-SEQ-MGR-OBJ)
                                               : set(event-obj) =

 let (out-events : set(event-obj) = {},
      clk-time : integer = get-import('simulation-time, subsystem, event-mgr),
      new-time : integer = 0)

%get the candidate new time from either the event-list or old-events depending
%on whether or not the event-list is empty

 (if Empty(ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) then
  (new-time <- Ev-Time(last(ED-SEQ-MGR-OBJ-OLD-EVENTS(event-mgr)``))
 else
  (new-time <- Ev-Time(ED-SEQ-MGR-OBJ-CURRENT-EVENT(event-mgr))));

   (if  ((clk-time < new-time) &
        ~(Transmit-Event-Obj(last(ED-SEQ-MGR-OBJ-OLD-EVENTS(event-mgr))) or
          Receive-Event-Obj(last(ED-SEQ-MGR-OBJ-OLD-EVENTS(event-mgr))))) then
      (out-events <- out-events union
                     set-export(subsystem, event-mgr,{set-attrs
                     (make-object('name-value-obj),
                                  'name-value-name, 'simulation-time,
                                  'name-value-value, new-time)});
       out-events <- {}; %% scrub out transmit events
       out-events <- gen-update-event('global-timer, [name(subsystem)])));
```

```
  out-events

%------------------------------
%-
%- Ed-Service-Event
%-
%- This is the routine where the current event is decoded and control
%- is passed to model components, or update events are scheduled for
%- other executive primitives which govern the clock or the passing of
%- data.
%-
%------------------------------

function Ed-Service-Event (subsystem : subsystem-obj,
                           event-mgr : ED-SEQ-MGR-OBJ) : set(event-obj) =

 format(debug-on, "ED-SEQ-MGR-OBJ-SERVICING event ~%~\\pp\\~%",
                  ED-SEQ-MGR-OBJ-CURRENT-EVENT(event-mgr));

 let (event-list : set(event-obj) = {},
      trash-events : set(event-obj) = {},
      target-subsys : subsystem-obj = undefined,
      bad-ones : seq(event-obj) = [],
      new-time : integer = 0,
      in-event : EVENT-OBJ = ED-SEQ-MGR-OBJ-CURRENT-EVENT(event-mgr))

%determine what subtype of event it is, and service it here or
%send and event to the controller to tell another executive
%primitive to service it.

% only remove one type of event object until this checks out o.k....
%remember what name-vale-obj does with multiple set states...

 Remove-Event-Obj(in-event)
               -->  (((event-type(in-event) = 'Transmit) -->
                       (bad-ones = [ e | (e : transmit-event-obj) &
                         (e in ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) &
                         (transmit-primitive-name(e) = object-name(in-event)) &
                         (transmit-subsystem-name(e) = first(subsystem-names(in-event)))]));

                     ((event-type(in-event) = 'Receive) -->
                       (bad-ones = [ e | (e : receive-event-obj) &
                         (e in ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) &
                         (object-name(e) = object-name(in-event)) &
                         (receive-import-name(e) = receive-import-name(in-event)) &
                         (subsystem-names(e) = subsystem-names(in-event))]));

                     ((event-type(in-event) = 'Update) -->
                       (bad-ones = [ e | (e : update-event-obj) &
                         (e in ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) &
                         (object-name(e) = object-name(in-event)) &
                         (subsystem-names(e) = subsystem-names(in-event))]));

                    ((event-type(in-event) = 'Set-State) -->
                       (bad-ones = [ e | (e : set-state-event-obj) &
                         (e in ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) &
```

```
                    (object-name(e) = object-name(in-event)) &
                    (subsystem-names(e) = subsystem-names(in-event))])));

%take all the events you find in bad ones and take them out of the event list.

 (enumerate Obj over ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr) do
  ((Obj in bad-ones) --> (Obj ~in ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))));

 Receive-Event-Obj(in-event) %if Rx then update conection manager to handle it
          --> (event-list = gen-update-event('conn-mgr, [name(subsystem)]));

 Start-Event-Obj(in-event)
          --> ((new-time <- Start-Time(in-event)) &
               (event-list = gen-update-event('global-timer, [name(subsystem)])));

 ~Start-Event-Obj(in-event)
         --> (new-time <- Ev-Time(in-event));


 Stop-Event-Obj(in-event)
         --> (event-list = {set-attrs(make-object('done-event-obj),
                                      'name, 'Done-1,
                                      'subsystem-names, [name(subsystem)],
                                      'object-name, [name(event-mgr)],
                                      'priority, 100,
                                      'ev-time, 0)}); %stop the simulation

 Transmit-Event-Obj(in-event)
         -->  (event-list = gen-update-event('conn-mgr, [name(subsystem)]));

%These transforms service new-data-events, update-events, and set-state-events

 ((New-Data-Event-Obj(in-event) or
  Update-Event-Obj(in-event) or Set-State-Event-Obj(in-event)) &
 (~empty(subsystem-names(in-event))))
       --> (Target-Subsys = find-object('subsystem-obj, first(subsystem-names(in-event))));

 (Defined?(Target-Subsys) &
 (New-Data-Event-Obj(in-event) or
  Update-Event-Obj(in-event) or Set-State-Event-Obj(in-event)))
       -->  ((Inevents(Target-Subsys) <- {in-event});
             (event-list <- execute-subsystem(Target-Subsys)));

%put the set-export transmit events in the trash

 trash-events <- trash-events union
              set-export(subsystem, event-mgr,
              {set-attrs(make-object('name-value-obj),
                'name-value-name, 'simulation-time,
                'name-value-value, new-time)});

 trash-events <- trash-events union
                set-export(subsystem, event-mgr,
                {set-attrs(make-object('name-value-obj),
                'name-value-name, 'curr-event,
                'name-value-value, ED-SEQ-MGR-OBJ-CURRENT-EVENT(event-mgr))});
```

```
%Delete the event from the current event-list
 Ed-Serve-Event(subsystem, in-event, event-mgr);


 event-list

%-----------------------------
%-
%- Ed-Serve-Event
%-
%- This mehtod removes an event from the object's event list after it has
%- been serviced.
%-
%-----------------------------

function Ed-Serve-Event (subsystem : subsystem-obj,
                         target-event : event-obj,
                         event-mgr  : ED-SEQ-MGR-OBJ) =


 ED-SEQ-MGR-OBJ-OLD-EVENTS(event-mgr) <- append(ED-SEQ-MGR-OBJ-OLD-EVENTS(event-mgr),
                                                Target-Event);

 (Target-Event in  ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr) -->
     Target-Event ~in ED-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))
```

A.8.2.4 Time-Driven Clock Manager.

```
!! in-package("AVSI")
!! in-grammar('user)

#||
Filename:td-clock.re
Author: Bob Welgan
Date: 6 Sep 93

Modifications:

Description:

This file describes a primitive object in the domain of application executives.
The primitive object definition supplied here conforms to the primitve object
template in Anderson's Appendix A.

The clock contains a time value and the methods necessary to change the time
and output it's value.
||#

% var TD-CLOCK-OBJ : object-class subtype-of Application-TD-Obj

 var TD-CLOCK-OBJ-INPUT-DATA   : set(import-obj) =
        {set-attrs (make-object('import-obj),
                    'import-name, 'new-time,
                    'import-category, 'time,
```

```
                    'import-type-data, 'integer)}

 var TD-CLOCK-OBJ-OUTPUT-DATA  : set(export-obj) =
       {set-attrs (make-object('export-obj),
                    'export-name, 'current-time,
                    'export-category, 'time,
                    'export-type-data, 'integer),

        set-attrs (make-object('export-obj),
                    'export-name, 'new-event,
                    'export-category, 'an-event,
                    'export-type-data, 'set-event-obj)}

 var TD-CLOCK-OBJ-COEFFICIENTS : map(TD-CLOCK-OBJ, set(name-value-obj))
  computed-using
    TD-CLOCK-OBJ-COEFFICIENTS(x) = {}

 var TD-CLOCK-OBJ-UPDATE-FUNCTION : map(TD-CLOCK-OBJ, symbol)
   computed-using
     TD-CLOCK-OBJ-UPDATE-FUNCTION(x) = 'TD-CLOCK-OBJ-SET-TD-CLOCK

% Other Attributes:
 var TD-CLOCK-OBJ-TIME : map(TD-CLOCK-OBJ, integer)
   computed-using
    TD-CLOCK-OBJ-TIME(x) = 0

 form Make-TD-CLOCK-Names-Unique
  unique-names-class('TD-CLOCK-OBJ, true)

%-----------------------------
%-
%- Function Set-TD-CLOCK
%-
%- This function sets the TD-CLOCK to the time indicated in the import area.
%-
%-----------------------------

function TD-CLOCK-OBJ-SET-TD-CLOCK (subsystem : subsystem-obj,
                                    the-clock : TD-CLOCK-OBJ) : set(event-obj) =

 format(debug-on, "TD-CLOCK-OBJ-SET on ~s~%", name(the-clock));

 let (time-value : integer = get-import('new-time, subsystem, the-clock),
      new-event  : event-obj = undefined,
      event-list : set(event-obj) = {})

 TD-CLOCK-OBJ-TIME(the-clock) <- time-value;
 format (true, "The current simulation time is  ~d~%", time-value);
 event-list <- set-export(subsystem, the-clock,
                          {set-attrs(make-object('name-value-obj),
                                                 'name-value-name, 'current-time,
                                                 'name-value-value, time-value)});
 new-event <- Arb(gen-transmit-event(name(subsystem),
                                     'current-time, name(the-clock)));

 Name(new-event) <- 'clock-update;
```

```
subsystem-names(new-event) <- [name(subsystem)];

event-list <- set-export(subsystem, the-clock,
                    {set-attrs(make-object('name-value-obj),
                                        'name-value-name, 'new-event,
                                        'name-value-value, {new-event})});

event-list <- {};
event-list
```

A.8.2.5 Time-Driven Sequential Event Manager.

```
!! in-package("AVSI")
!! in-grammar('user)

#||
Filename: td-seq-event-man.re
Author: Bob Welgan
Date: 6 Sep 93

Modifications: Fixed dual clock updates on start (30 Oct RLW)
               Passes copy of application event to model subsystem (22 Nov RLW)

Description:

This file describes a primitive object in the domain of application executives.
The primitive object definition supplied here conforms to the primitve object
template on page 2, Appendix A, Anderson's thesis.

The schedule contains a list of events and the methods necessary to manipulate
the list.
||#

% var TD-SEQ-MGR-OBJ            : object-class subtype-of Application-Exec-Obj

 var TD-SEQ-MGR-OBJ-INPUT-DATA     : set(import-obj) =
       {set-attrs (make-object('import-obj),
                   'import-name, 'simulation-time,
                   'import-category, 'time,
                   'import-type-data, 'integer),

        set-attrs (make-object('import-obj),
                   'import-name, 'new-event,
                   'import-category, 'an-event,
                   'import-type-data, 'set-event-obj)}

 var TD-SEQ-MGR-OBJ-OUTPUT-DATA     : set(export-obj) =
      {set-attrs (make-object('export-obj),
                  'export-name, 'curr-event,
                  'export-category, 'an-event,
                  'export-type-data, 'event-obj),

       set-attrs (make-object('export-obj),
                  'export-name, 'simulation-time,
                  'export-category, 'time,
```

```
                    'export-type-data, 'integer)}

 var TD-SEQ-MGR-OBJ-COEFFICIENTS : map(TD-SEQ-MGR-OBJ, set(name-value-obj))
   computed-using
     TD-SEQ-MGR-OBJ-COEFFICIENTS(x) = {}

 var TD-SEQ-MGR-OBJ-UPDATE-FUNCTION : map(TD-SEQ-MGR-OBJ, symbol)
   computed-using
     TD-SEQ-MGR-OBJ-UPDATE-FUNCTION(x) = 'TD-SEQ-MGR-OBJ-UPDATE1

%Td-Seq-Mgr Object Attributes

 var TD-SEQ-MGR-OBJ-EVENT-LIST : map(TD-SEQ-MGR-OBJ, seq(EVENT-OBJ))
   computed-using
     TD-SEQ-MGR-OBJ-EVENT-LIST(x) = []

 var TD-SEQ-MGR-OBJ-OLD-EVENTS : map(TD-SEQ-MGR-OBJ, seq(EVENT-OBJ))
   computed-using
     TD-SEQ-MGR-OBJ-OLD-EVENTS(x) = []

 var TD-SEQ-MGR-OBJ-CURRENT-EVENT : map(TD-SEQ-MGR-OBJ, EVENT-OBJ)
   computed-using
     TD-SEQ-MGR-OBJ-CURRENT-EVENT(x) = first(TD-SEQ-MGR-OBJ-EVENT-LIST(x))

 form Make-TD-SEQ-MGR-Names-Unique
  unique-names-class('TD-SEQ-MGR-OBJ, true)

%-----------------------------
%-
%- Time-Driven-Sequential-Manager-Object Update Function
%-
%- This rather large primitive update function is broken into three different
%- steps.  The update begins by servicing the current event.  Then, it calls
%- a function to determine if the event needs to be rescheduled at a new time.
%-
%-----------------------------

function TD-SEQ-MGR-OBJ-UPDATE1 (subsystem : subsystem-obj,
                                 event-mgr  : TD-SEQ-MGR-OBJ) : set(event-obj) =

  format (debug-on, "Events are now being serviced by ~s~%", name(event-mgr));

  let (exec-events : set(event-obj) = {},
       new-events  : set(event-obj) = {},
       current-time : Integer = get-import('simulation-time, subsystem, event-mgr),
       Running : Boolean = (~Empty(TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) &
                  ~Stop-Event-Obj(last(TD-SEQ-MGR-OBJ-OLD-EVENTS(event-mgr)))),
       New-Ones : Any-Type = 'Undefined,
       Really-New : set(event-obj) = {},
       Serviced-An-Event : Boolean = False,
       Done : Boolean = False)

 (if Running then
     New-Ones <- get-import('new-event, subsystem, event-mgr);
     format(debug-on, "New-Ones are ~A~%", New-Ones);
     % check the import for new, valid events from other primitives...
```

```
   (New-Ones ~= 0) -->
     ((if Event-Obj(New-Ones) then
       (Really-New <- Really-New with New-Ones)
       else Really-New <- New-Ones));
     (enumerate Obj over Really-New do
       (if (Event-Obj(Obj) & ~New-Data-Event-Obj(Obj) &
            (Obj ~in TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) &
            (Obj ~in TD-SEQ-MGR-OBJ-OLD-EVENTS(event-mgr))) then
         ((new-events <- new-events with Obj);
         (format(debug-on, "Event Manager adding this event from import ~A~%",
                         Obj)))));

    % if any new events are valid, add them...
    ~Empty(new-events)
        --> (Td-Add-Events(subsystem, event-mgr, new-events));

   %determine if an event should be serviced now...

   ((Ev-Time(TD-SEQ-MGR-OBJ-CURRENT-EVENT(event-mgr)) = current-time)
        -->  ((new-events <- Td-Service-Event(subsystem, event-mgr)) &
             Serviced-An-Event));

   Serviced-An-Event
       -->  ((enumerate Obj over new-events do    %scrub out new-data events
              (New-Data-Event-Obj(Obj) --> Obj ~in new-events));
             %determine if this is done
             (enumerate Obj over new-events do
               (Done-Event-Obj(Obj) --> (Done &
                                         (exec-events <- new-events))));

             Schedule-New-Event(subsystem, event-mgr)); % re-schedule

  ~Done  -->
    (exec-events <- exec-events union {e | (e :event-obj) &
                                    (e in new-events) &
                                    (first(subsystem-names(e)) = 'app-exec)};
   new-events <- setdiff(new-events, exec-events);
   Td-Add-Events(subsystem, event-mgr, new-events)); %add events raistd by app

%if this hasn't serviced an event, then must make the clock tick

   ~Serviced-an-Event
        --> exec-events <- exec-events union Increment-Clock(subsystem,
                                                             event-mgr));
Serviced-An-Event <- False;

~Running
    --> (exec-events <- {set-attrs(make-object('done-event-obj),
                                   'name, 'Done-2,
                                   'subsystem-names, [name(subsystem)],
                                   'object-name, [name(event-mgr)],
                                   'priority, 100,
                                   'ev-time, 0)});

(enumerate Obj over TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr) do
```

```
    format(debug-on, "Event in Event-List ~A~%", Obj));

  exec-events

%------------------------------
%-
%- Td-Add-Events
%-
%- This mehtod gets all the events raised as a result of application executive
%- subsystem execution and places them in the event-mgr.
%-
%------------------------------

function Td-Add-Events (subsystem : subsystem-obj,
                        event-mgr  : TD-SEQ-MGR-OBJ,
                        in-events  : set(event-obj)) =

 let (Done  : Boolean = False,
      index : Integer = 1,
      Sim-Time  : Integer = get-import('simulation-time, subsystem, event-mgr))

  format (debug-on, "New Event Added To Event-Mgr ~A~%",in-events);

  %add the current absolute time to each event's relative time in the
  %set of incoming events

  (enumerate new-event over in-events do
    (Ev-Time(new-event) <- (Ev-Time(new-event) + Sim-Time)));

  %take each event object and place it in the event-mgr one at a time
  %the event must either go at the head of the sequence, in the middle,
  %or at the end.
  %find out where it should go...

 (enumerate new-event over in-events do
   (while ((index <= size(TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))) & ~Done) do

    ((Ev-Time(new-event) < Ev-Time((TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))(index)))
       --> (Done);

     (Ev-Time(new-event) > Ev-Time((TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))(index)))
       --> (index = (index + 1));

     ((Ev-Time(new-event) = Ev-Time((TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))(index))) &
     (Priority(new-event) <= Priority((TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))(index))))
       --> (index = (index + 1));

     ((Ev-Time(new-event) = Ev-Time((TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))(index))) &
     (Priority(new-event) > Priority((TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))(index))))
       --> (Done))); %end while

  %Place the new event object at the location pointed to by index

     TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)
           <- insert(TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr),index, new-event);
     Done <- False;
```

```
    index <- 1)

%------------------------------
%-
%- Increment-Clock
%-
%- This makes the clock tick by sending a new time one time unit greater than
%- the current time to the new-time export object and scheduling an update
%- event for the executive.
%-
%------------------------------

function Increment-Clock (subsystem : subsystem-obj,
                          event-mgr : TD-SEQ-MGR-OBJ)  : set(event-obj) =

 let (out-events : set(event-obj) = {},
      clk-time : integer = get-import('simulation-time, subsystem, event-mgr),
      new-time : integer = 0)

  new-time <- clk-time + 1;  % note: delay delta is one...

  (out-events <- out-events union
                 set-export(subsystem, event-mgr,{set-attrs
                 (make-object('name-value-obj),
                              'name-value-name, 'simulation-time,
                              'name-value-value,
                               new-time)});
   out-events <- {}; %% scrub out events raised by set-export
   out-events <- gen-update-event('global-timer, [name(subsystem)]));

  out-events

%------------------------------
%-
%- Td-Service-Event
%-
%- This is the routine where the current event is decoded and control
%- is passed to model components, or update events are scheduled for
%- other executive primitives which govern the clock or the passing of
%- data.
%-
%------------------------------

function Td-Service-Event (subsystem : subsystem-obj,
                           event-mgr : TD-SEQ-MGR-OBJ) : set(event-obj) =

 format(debug-on, "TD-SEQ-MGR-OBJ-SERVICING event ~%~\\pp\\~%",
                  TD-SEQ-MGR-OBJ-CURRENT-EVENT(event-mgr));

 let (event-list : set(event-obj) = {},
      trash-events : set(event-obj) = {},
      target-subsys : subsystem-obj = undefined,
      bad-ones : seq(event-obj) = [],
      new-time : integer = 0,
      in-event : EVENT-OBJ = TD-SEQ-MGR-OBJ-CURRENT-EVENT(event-mgr))
```

```
%determine what subtype of event it is, and service it here or
%send and event to the controller to tell another executive
%primitive to service it.

% only remove one type of event object until this checks out o.k....
%remember what name-vale-obj does with multiple set states...

 Remove-Event-Obj(in-event)
                        -->  (((event-type(in-event) = 'Transmit) -->
                          (bad-ones = [ e | (e : transmit-event-obj) &
                            (e in TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) &
                            (transmit-primitive-name(e) = object-name(in-event)) &
                            (transmit-subsystem-name(e) = first(subsystem-names(in-event)))]));

                         ((event-type(in-event) = 'Receive) -->
                          (bad-ones = [ e | (e : receive-event-obj) &
                            (e in TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) &
                            (object-name(e) = object-name(in-event)) &
                            (receive-import-name(e) = receive-import-name(in-event)) &
                            (subsystem-names(e) = subsystem-names(in-event))]));

                         ((event-type(in-event) = 'Update) -->
                          (bad-ones = [ e | (e : update-event-obj) &
                            (e in TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) &
                            (object-name(e) = object-name(in-event)) &
                            (subsystem-names(e) = subsystem-names(in-event))]));

                        ((event-type(in-event) = 'Set-State) -->
                          (bad-ones = [ e | (e : set-state-event-obj) &
                            (e in TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr)) &
                            (object-name(e) = object-name(in-event)) &
                            (subsystem-names(e) = subsystem-names(in-event))]));

 %take all the events you find in bad ones and take them out of the event list.

 (enumerate Obj over TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr) do
  ((Obj in bad-ones) --> (Obj ~in TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))));

 Receive-Event-Obj(in-event) %if Rx then update conection manager to handle it
           --> (event-list = gen-update-event('conn-mgr, [name(subsystem)]));

 Start-Event-Obj(in-event)
           --> ((new-time = Start-Time(in-event)) &
               (event-list = gen-update-event('global-timer, [name(subsystem)])));
 %%%%Put another transform here to set the export to start-time on Start-Event
 Start-Event-Obj(in-event)
          --> (trash-events = trash-events union          %put the Tx events
              set-export(subsystem, event-mgr,           %made by set-export
              {set-attrs(make-object('name-value-obj),    %in the trash
                'name-value-name, 'simulation-time,
                'name-value-value, new-time)}));


 Stop-Event-Obj(in-event)
          --> (event-list = {set-attrs(make-object('done-event-obj),
                                       'name, 'Done-1,
```

```
                                        'subsystem-names, [name(subsystem)],
                                        'object-name, [name(event-mgr)],
                                        'priority, 100,
                                        'ev-time, 0)}); %stop the simulation

 Transmit-Event-Obj(in-event)
          --> (event-list = gen-update-event('conn-mgr, [name(subsystem)]));

%These transforms service new-data-events, update-events, and set-state-events

 ((New-Data-Event-Obj(in-event) or
  Update-Event-Obj(in-event) or Set-State-Event-Obj(in-event)) &
 (~empty(subsystem-names(in-event))))
        --> (Target-Subsys = find-object('subsystem-obj, first(subsystem-names(in-event))));

 (Defined?(Target-Subsys) &
 (New-Data-Event-Obj(in-event) or
  Update-Event-Obj(in-event) or Set-State-Event-Obj(in-event)))
        --> ((Inevents(Target-Subsys) <- {copy-term(in-event)});
             (event-list <- execute-subsystem(Target-Subsys)));

%put the set-export transmit events in the trash

 trash-events <- trash-events union
                 set-export(subsystem, event-mgr,
                 {set-attrs(make-object('name-value-obj),
                  'name-value-name, 'curr-event,
                  'name-value-value, TD-SEQ-MGR-OBJ-CURRENT-EVENT(event-mgr))});

%remove this event from the event list
 Td-Serve-Event(subsystem, in-event, event-mgr);

 event-list

%-----------------------------
%-
%- Schedule-New-Event
%-
%- This mehtod determines if the current event needs to be rescheduled at a
%- later time, and what that later time should be.
%-
%-----------------------------

function Schedule-New-Event (subsystem : subsystem-obj,
                             event-mgr  : TD-SEQ-MGR-OBJ) =

(format (factive, "Entering the function Schedule-New-Event in TD-EXEC"));

let (event : event-obj = (last(TD-SEQ-MGR-OBJ-OLD-EVENTS(event-mgr))),
     new-event : event-obj = undefined,
     Done : boolean = false,
     time-delta : integer = 1)

 format(debug-on, "Reschedule Event testing ~A~%", event);

 Update-Event-Obj(event)
```

```
  --> (new-event <- make-object('update-event-obj);
       name(new-event) <- name(event);
       priority(new-event) <- priority(event);
       object-name(new-event) <- object-name(event);
       subsystem-names(new-event) <- subsystem-names(event);
       while ~Done do
         (~ex (x) (x in TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr) &
                  (ev-time(x) = ev-time(event) + time-delta))
                      --> (ev-time(new-event) <- time-delta;
                           Td-Add-Events(subsystem, event-mgr, {new-event});
                           Done <- True);

            ex (x) (x in TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr) &
                   (ev-time(x) = ev-time(event) + time-delta) &
                   (first(subsystem-names(x)) = first(subsystem-names(new-event)) or
                    (first(subsystem-names(x)) = 'app-exec)) &
                    ~Done)
                      --> (ev-time(new-event) <- time-delta;
                           Td-Add-Events(subsystem, event-mgr, {new-event});
                           Done <- True);
            time-delta <- time-delta + 1))


%-----------------------------
%-
%- Td-Serve-Event
%-
%- This mehtod removes an event from the object's event list after it has
%- been serviced.
%-
%-----------------------------

function Td-Serve-Event (subsystem : subsystem-obj,
                         target-event : event-obj,
                         event-mgr  : TD-SEQ-MGR-OBJ) =


 TD-SEQ-MGR-OBJ-OLD-EVENTS(event-mgr) <- append(TD-SEQ-MGR-OBJ-OLD-EVENTS(event-mgr),
                                                Target-Event);

 (Target-Event in  TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr) -->
     Target-Event ~in TD-SEQ-MGR-OBJ-EVENT-LIST(event-mgr))
```

A.9 Summary

The domain analysis process defined in Chapter 3 resulted in both an informal domain model and a formal domain model of an application executive. The informal portion consisted of Rumbaugh object and dynamic models. The formal portion consisted of Architect-OCU compliant primitives, written in the REFINE wide- spectrum language.

Appendix B. Test Cases and Results

B.1 Introduction

This appendix contains a representative set of the test cases which validated the operations of the Architect application executive in the event-driven sequential and time-driven sequential domains. The complete set of test cases and results for the Architect application executive is available upon request from:

Maj Paul Bailor
AFIT/ENG
2950 P Street
Wright-Patterson AFB, OH 45433-7765

(513)255-9263
DSN 785-9263
email: pbailor@afit.af.mil

B.2 Event-Driven Sequential Sample Test

Three applications, expressed in the OCU architecture-specific language, (of which the application executive domain specific language is a subset) tested the application executive in the event-driven sequential mode of execution. The test script from the one-subsystem test is presented below. This test is designed to show that the executive can control one subsystem, accept events from the subsystem, and include them on the event list.

```
%
% filename: simple-app
% author  : Bob Welgan
% date    : 30 Sep 93
%
% This file, written in OCU language, is designed to be parsed into
% the Refine object base.  It defines a complete application: model
% subsystems and the event-driven sequential application executive
% subsystem.  This application is desinged to exercise the application
% executive.


application definition test-1-app

  execution-mode: event-driven-sequential
```

```
% define appliction subsystems here...

  switch sw-1
  delay: 0
  is debounced
  manufacturer: "Riddler"
  position: off

  switch sw-2
  delay: 0
  is debounced
  manufacturer: "Riddler"
  position: off

  and-gate and-1
  delay: 5
  fan-out: 5
  is mil-spec
  manufacturer: "BOBCO"
  power level: 5.0

  led led-1

  subsystem sub-1 is

    controls: sw-1, sw-2, and-1, led-1

    imports: IN1 SIGNAL BOOLEAN NIL AND-1
             (OUT1 SUB-1 SW-1)
             import-path: (SUB-1)
             import-owner:  SUB-1 not-changed

             IN2 SIGNAL BOOLEAN NIL AND-1
             (OUT1 SUB-1 SW-2)
             import-path: (SUB-1)
             import-owner:  SUB-1 not-changed

             IN1 SIGNAL BOOLEAN NIL LED-1
             (OUT1 SUB-1 AND-1)
             import-path: (SUB-1)
             import-owner:  SUB-1 not-changed

    exports: OUT1 SIGNAL BOOLEAN NIL AND-1
             export-path: (SUB-1)
             export-owner: SUB-1

             OUT1 SIGNAL BOOLEAN NIL SW-1
             export-path: (SUB-1)
             export-owner: SUB-1

             OUT1 SIGNAL BOOLEAN NIL SW-2
             export-path: (SUB-1)
             export-owner: SUB-1

    update procedure:
      update sw-1
```

```
      update sw-2
      update and-1
      update led-1

% define application executive
  event-driven-clock  global-timer
    is-at-time: 0

  event-driven-sequential-event-manager  event-handler
    manages:  start-event event1
              ev-time: 0
              for-primitive: event-handler
              through-subsystems: app-exec
              priority: 100
              start-time: 1,

              set-state-event event2
              ev-time: 1
              for-primitive: sw-1
              attr-values: (position, on)
              through-subsystems: sub-1
              priority: 1,

              set-state-event event3
              ev-time: 1
              for-primitive: sw-2
              attr-values: (position, on)
              through-subsystems: sub-1
              priority: 1,

              stop-event event4
              ev-time: 50
              for-primitive: event-handler
              through-subsystems: app-exec
              priority: 100
              stop-time: 50

  connection-manager conn-mgr
    manages: connection con-0

  subsystem app-exec is

    controls: global-timer, event-handler, conn-mgr

    imports: SIMULATION-TIME TIME INTEGER 0 EVENT-HANDLER
             (CURRENT-TIME APP-EXEC GLOBAL-TIMER)
             import-path: (APP-EXEC)
             import-owner:  APP-EXEC is-changed

             NEW-EVENT AN-EVENT SET-EVENT-OBJ 0 EVENT-HANDLER
             (NEW-EVENT APP-EXEC CONN-MGR)
             import-path:  (APP-EXEC)
             import-owner:  APP-EXEC is-changed

             NEW-TIME TIME INTEGER 0 GLOBAL-TIMER
             (SIMULATION-TIME APP-EXEC EVENT-HANDLER)
```

```
            import-path:  (APP-EXEC)
            import-owner:  APP-EXEC is-changed

            CURR-EVENT AN-EVENT EVENT-OBJ 0 CONN-MGR
            (CURR-EVENT APP-EXEC EVENT-HANDLER)
            import-path: (APP-EXEC)
            import-owner: APP-EXEC is-changed

   exports: SIMULATION-TIME TIME INTEGER 0 EVENT-HANDLER
            export-path: (APP-EXEC)
            export-owner: APP-EXEC

            CURR-EVENT AN-EVENT EVENT-OBJ 0 EVENT-HANDLER
            export-path: (APP-EXEC)
            export-owner: APP-EXEC

            CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
            export-path: (APP-EXEC)
            export-owner: APP-EXEC

            NEW-EVENT AN-EVENT SET-EVENT-OBJ 0 CONN-MGR
            export-path: (APP-EXEC)
            export-owner: APP-EXEC

   update procedure:
     update global-timer
     update event-handler
     update conn-mgr

application test-me is
   controls: app-exec, sub-1
   update procedure:
     update app-exec
     update sub-1
```

Here are the results of the test:

```
(* This is the state of the executive prior to execution *)
.> (pn 'app-exec)
subsystem APP-EXEC is controls:
  GLOBAL-TIMER, EVENT-HANDLER, CONN-MGR
  imports:
    SIMULATION-TIME TIME INTEGER 0 EVENT-HANDLER
      ( CURRENT-TIME APP-EXEC GLOBAL-TIMER
      ) import-path: ( APP-EXEC ) import-owner: APP-EXEC
          is-changed
      NEW-EVENT AN-EVENT SET-EVENT-OBJ 0 EVENT-HANDLER
        ( NEW-EVENT APP-EXEC CONN-MGR
        ) import-path: ( APP-EXEC ) import-owner: APP-EXEC
            is-changed
      NEW-TIME TIME INTEGER 0 GLOBAL-TIMER
        ( SIMULATION-TIME APP-EXEC EVENT-HANDLER
        ) import-path: ( APP-EXEC ) import-owner: APP-EXEC
            is-changed
      CURR-EVENT AN-EVENT EVENT-OBJ 0 CONN-MGR
        ( CURR-EVENT APP-EXEC EVENT-HANDLER
        ) import-path: ( APP-EXEC ) import-owner: APP-EXEC
```

```
          is-changed
  exports:
    SIMULATION-TIME TIME INTEGER 0 EVENT-HANDLER
      export-path: ( APP-EXEC ) export-owner: APP-EXEC
      CURR-EVENT AN-EVENT EVENT-OBJ 0 EVENT-HANDLER
        export-path: ( APP-EXEC ) export-owner: APP-EXEC
      CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
        export-path: ( APP-EXEC ) export-owner: APP-EXEC
      NEW-EVENT AN-EVENT SET-EVENT-OBJ 0 CONN-MGR
        export-path: ( APP-EXEC ) export-owner: APP-EXEC
  initialize procedure: update procedure:
  update GLOBAL-TIMER update EVENT-HANDLER update CONN-MGR
.> (pn 'event-handler)
event-driven-sequential-event-manager EVENT-HANDLER
  manages:
  start-event EVENT1 ev-time: 0 for-primitive: EVENT-HANDLER
    through-subsystems: APP-EXEC priority: 100 start-time: 1,
  set-state-event EVENT2 ev-time: 1 for-primitive: SW-1
    attr-values: (POSITION, ON) through-subsystems: SUB-1
    priority: 1,
  set-state-event EVENT3 ev-time: 1 for-primitive: SW-2
    attr-values: (POSITION, ON) through-subsystems: SUB-1
    priority: 1,
  stop-event EVENT4 ev-time: 50 for-primitive: EVENT-HANDLER
    through-subsystems: APP-EXEC priority: 100 stop-time: 50
.> (ar 16)
Rule successfully applied.
application definition TEST-1-APP
  execution-mode: EVENT-DRIVEN-SEQUENTIAL SW-1 SW-2 AND-1
    LED-1 SUB-1 GLOBAL-TIMER EVENT-HANDLER CONN-MGR APP-EXEC
    TEST-ME
.> (ar 15)
The current simulation time is  1  (* Start time is 1 *)
scavenging...done
The current simulation time is  6
LED LED-1 = ON                    (* The LED lit at time 6 *)
The current simulation time is  50

 Execution Stopped
Rule successfully applied.
application definition TEST-1-APP
  execution-mode: EVENT-DRIVEN-SEQUENTIAL SW-1 SW-2 AND-1
    LED-1 SUB-1 GLOBAL-TIMER EVENT-HANDLER CONN-MGR APP-EXEC
    TEST-ME
.> (ar 18)
Rule successfully applied.
application definition TEST-1-APP
  execution-mode: EVENT-DRIVEN-SEQUENTIAL SW-1 SW-2 AND-1
    LED-1 SUB-1 GLOBAL-TIMER EVENT-HANDLER CONN-MGR APP-EXEC
    TEST-ME

(* This is the Old-Events List after execution *)
.> (ar 19)
start-event EVENT1 ev-time: 0 for-primitive: EVENT-HANDLER
  through-subsystems: APP-EXEC priority: 100 start-time: 1
```

```
set-state-event EVENT2 ev-time: 1 for-primitive: SW-1
  attr-values: (POSITION, ON) through-subsystems: SUB-1
  priority: 1

set-state-event *UNDEFINED* ev-time: 1 for-primitive:
  SW-1 attr-values: (OUT1, T) through-subsystems: SUB-1
  priority: 5

transmit-event *UNDEFINED* ev-time: 1 from-primitive:
  SW-1 in-subsystem: SUB-1 from-export: OUT1 priority: 50

set-state-event EVENT3 ev-time: 1 for-primitive: SW-2
  attr-values: (POSITION, ON) through-subsystems: SUB-1
  priority: 1

set-state-event *UNDEFINED* ev-time: 1 for-primitive:
  SW-2 attr-values: (OUT1, T) through-subsystems: SUB-1
  priority: 5

transmit-event *UNDEFINED* ev-time: 1 from-primitive:
  SW-2 in-subsystem: SUB-1 from-export: OUT1 priority: 50

update-event *UNDEFINED* ev-time: 1 for-primitive: AND-1
  through-subsystems: SUB-1 priority: 1

set-state-event *UNDEFINED* ev-time: 6 for-primitive:
  AND-1 attr-values: (OUT1, T) through-subsystems: SUB-1
  priority: 5

transmit-event *UNDEFINED* ev-time: 6 from-primitive:
  AND-1 in-subsystem: SUB-1 from-export: OUT1 priority: 50

update-event *UNDEFINED* ev-time: 6 for-primitive: LED-1
  through-subsystems: SUB-1 priority: 1

set-state-event *UNDEFINED* ev-time: 6 for-primitive:
  LED-1 attr-values: (STATE, ON) through-subsystems: SUB-1
  priority: 5

stop-event EVENT4 ev-time: 50 for-primitive: EVENT-HANDLER
  through-subsystems: APP-EXEC priority: 100 stop-time: 50

Rule successfully applied.
application definition TEST-1-APP
  execution-mode: EVENT-DRIVEN-SEQUENTIAL SW-1 SW-2 AND-1
    LED-1 SUB-1 GLOBAL-TIMER EVENT-HANDLER CONN-MGR APP-EXEC
    TEST-ME
.>
```

B.3 Time-Driven Sequential Sample Test

Three applications tested Architect application executive time-driven sequential operation. This exercised the executive's ability to control the flow of data between two, top-level subsystems in time-driven sequential mode.

```
%
% filename: two-sub-test
% author  : Bob Welgan
% date    : 4 Nov 93
%
% This file, written in OCU language, is designed to be parsed into
% the Refine object base.  It defines the application executive
% subsystem.  Although it is saved as an application definition, it
% does not define an entire application.  The user must insert model
% components from a different domain into the Refine object base and
% link the model to the executive subsystem defined here.
%


application definition td-test

  execution-mode: time-driven-sequential
  connected-by
  connection con-0
             links: CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
                    export-path: (APP-EXEC GLOBAL-TIMER)
                    export-owner: APP-EXEC

            to-imp: SIM-TIME TIME INTEGER 0 the-tank
                    ( )
                    import-path: (SUB-2)
                    import-owner: SUB-2 not-changed
         with-data: NIL
  connection-state: NOT-CONSUMED

  connection con-1
             links: CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
                    export-path: (APP-EXEC GLOBAL-TIMER)
                    export-owner: APP-EXEC

            to-imp: SIM-TIME TIME INTEGER 0 the-throttle
                    ( )
                    import-path: (SUB-2)
                    import-owner: SUB-2 not-changed
         with-data: NIL
  connection-state: NOT-CONSUMED

  connection con-2
             links: CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
                    export-path: (APP-EXEC GLOBAL-TIMER)
                    export-owner: APP-EXEC

            to-imp: SIM-TIME TIME INTEGER 0 the-engine
                    ( )
                    import-path: (SUB-2)
                    import-owner: SUB-2 not-changed
         with-data: NIL
  connection-state: NOT-CONSUMED

  connection con-3
             links: OUT1 SIGNAL BOOLEAN NIL ENGINE-START
```

```
                export-path: (SUB-1)
                export-owner: SUB-1

        to-imp: START SIGNAL BOOLEAN NIL the-engine
                ( )
                import-path: (SUB-2)
                import-owner: SUB-2 not-changed
      with-data: NIL
connection-state: NOT-CONSUMED

connection con-4
         links: OUT1 SIGNAL BOOLEAN NIL FUEL-PUMP-START
                export-path: (SUB-1)
                export-owner: SUB-1

        to-imp: START SIGNAL BOOLEAN NIL the-tank
                ( )
                import-path: (SUB-2)
                import-owner: SUB-2 not-changed
      with-data: NIL
connection-state: NOT-CONSUMED

% subsystem definition begins here ....

fuel-tank the-tank
capacity: 100.0 % pounds
empty weight: 50.0
fuel level: 100.0
flow rate: 0.0  % pounds/sec
pump off
last time: 0
fuel density: 1.0


throttle the-throttle
max flow rate: 0.2 % pounds/sec


jet engine the-engine
thrust factor: 1000.0
max flow rate: 0.15 % pounds/sec
mode: off

switch engine-start
delay: 0
is debounced
manufacturer: "Riddler"
position: off

switch fuel-pump-start
delay: 0
is debounced
manufacturer: "Riddler"
position: off

subsystem  sub-1 is
```

```
  controls: engine-start, fuel-pump-start

  exports: OUT1 SIGNAL BOOLEAN NIL ENGINE-START
           export-path: (SUB-1)
           export-owner: SUB-1

           OUT1 SIGNAL BOOLEAN NIL FUEL-PUMP-START
           export-path: (SUB-1)
           export-owner: SUB-1

  update procedure:
    update engine-start
    update fuel-pump-start


subsystem  sub-2 is

  controls: the-tank, the-throttle, the-engine

  imports: START SIGNAL BOOLEAN NIL the-tank
           ( )
           import-path: (SUB-2)
           import-owner: SUB-2 not-changed

           START SIGNAL BOOLEAN NIL the-engine
           ( )
           import-path: (SUB-2)
           import-owner: SUB-2 not-changed

           THROTTLE-INDEX PERCENT REAL 0.25 the-throttle
           ( )
           import-path: (SUB-2)
           import-owner: SUB-2 not-changed

           CONSUMPTION-RATE FLOW-RATE REAL 0.0 the-tank
            (FUEL-FLOW-RATE SUB-2 the-engine)
           import-path: (SUB-2)
           import-owner: SUB-2 not-changed

           FUEL-AVAILABLE? SIGNAL BOOLEAN NIL the-throttle
            (FUEL-AVAILABLE? SUB-2 the-tank)
           import-path: (SUB-2)
           import-owner: SUB-2 not-changed

           INFLOW-RATE FLOW-RATE REAL 0.0 the-engine
            (REQUESTED-FLOW-RATE SUB-2 the-throttle)
           import-path: (SUB-2)
           import-owner: SUB-2 not-changed

           SIM-TIME TIME INTEGER 0 the-tank
           ( )
           import-path: (SUB-2)
           import-owner: SUB-2 not-changed

           SIM-TIME TIME INTEGER 0 the-throttle
```

```
            ( )
            import-path: (SUB-2)
            import-owner: SUB-2 not-changed

            SIM-TIME TIME INTEGER 0 the-engine
            ( )
            import-path: (SUB-2)
            import-owner: SUB-2 not-changed

    exports: FUEL-AVAILABLE? SIGNAL BOOLEAN NIL the-tank
            export-path: (SUB-2)
            export-owner: SUB-2

            FUEL-TANK-WEIGHT FORCE REAL 150.0 the-tank
            export-path: (SUB-2)
            export-owner: SUB-2

            REQUESTED-FLOW-RATE FLOW-RATE REAL 0.0 the-throttle
            export-path: (SUB-2)
            export-owner: SUB-2

            THRUST FORCE REAL 0.0 the-engine
            export-path: (SUB-2)
            export-owner: SUB-2

            FUEL-FLOW-RATE FLOW-RATE REAL 0.0 the-engine
            export-path: (SUB-2)
            export-owner: SUB-2

   update procedure:
     update the-tank
     update the-throttle
     update the-engine


% application executive definition begins here ..
  time-driven-clock  global-timer
    is-at-time: 0

  time-driven-sequential-event-manager  event-handler
     manages: start-event event1
            ev-time: 0
            for-primitive: event-handler
            through-subsystems: app-exec
            priority: 100
            start-time: 0,

% put application events here .....

            update-event event2
            ev-time: 1
            for-primitive: the-tank
            through-subsystems: sub-2
            priority: 1,

            update-event event3
```

```
            ev-time: 1
            for-primitive: the-throttle
            through-subsystems: sub-2
            priority: 1,

            update-event event4
            ev-time: 1
            for-primitive: the-engine
            through-subsystems: sub-2
            priority: 1,

            set-state-event event5
            ev-time: 3
            for-primitive: engine-start
            attr-values: (position, on)
            through-subsystems: sub-1
            priority: 1,

            set-state-event event6
            ev-time: 3
            for-primitive: fuel-pump-start
            attr-values: (position, on)
            through-subsystems: sub-1
            priority: 1,

            stop-event event3
            ev-time: 9
            for-primitive: event-handler
            through-subsystems: app-exec
            priority: 100
            stop-time: 9

connection-manager conn-mgr
  manages: connection con-0

subsystem app-exec is

  controls: global-timer, event-handler, conn-mgr

  imports: SIMULATION-TIME TIME INTEGER 0 EVENT-HANDLER
           (CURRENT-TIME APP-EXEC GLOBAL-TIMER)
           import-path: (APP-EXEC EVENT-HANDLER)
           import-owner:  APP-EXEC is-changed

           NEW-EVENT AN-EVENT SET-EVENT-OBJ 0 EVENT-HANDLER
           (NEW-EVENT APP-EXEC CONN-MGR,
            NEW-EVENT APP-EXEC GLOBAL-TIMER)
           import-path:  (APP-EXEC EVENT-HANDLER)
           import-owner:  APP-EXEC is-changed

           NEW-TIME TIME INTEGER 0 GLOBAL-TIMER
           (SIMULATION-TIME APP-EXEC EVENT-HANDLER)
           import-path:  (APP-EXEC GLOBAL-TIMER)
           import-owner:  APP-EXEC is-changed

           CURR-EVENT AN-EVENT EVENT-OBJ 0 CONN-MGR
```

```
            (CURR-EVENT APP-EXEC EVENT-HANDLER)
            import-path: (APP-EXEC CONN-MGR)
            import-owner: APP-EXEC is-changed

    exports: SIMULATION-TIME TIME INTEGER 0 EVENT-HANDLER
            export-path: (APP-EXEC EVENT-HANDLER)
            export-owner: APP-EXEC

            CURR-EVENT AN-EVENT EVENT-OBJ 0 EVENT-HANDLER
            export-path: (APP-EXEC EVENT-HANDLER)
            export-owner: APP-EXEC

            CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
            export-path: (APP-EXEC GLOBAL-TIMER)
            export-owner: APP-EXEC

            NEW-EVENT AN-EVENT SET-EVENT-OBJ 0 CONN-MGR
            export-path: (APP-EXEC CONN-MGR)
            export-owner: APP-EXEC

            NEW-EVENT AN-EVENT SET-EVENT-OBJ 0 GLOBAL-TIMER
            export-path: (APP-EXEC GLOBAL-TIMER)
            export-owner: APP-EXEC

    update procedure:
      update global-timer
      update event-handler
      update conn-mgr

 application test-me is
    controls: app-exec, sub-1, sub-2
    update procedure:
      update app-exec
      update sub-1
      update sub-2
```

Here are the results of a test using this application:

```
(* This is the application state prior to execution *)
.> (mcn 'td-test)
application definition TD-TEST
  execution-mode: TIME-DRIVEN-SEQUENTIAL
    connected-by
      connection CON-0
        links:
          CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
            export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
              APP-EXEC
          to-imp:
          SIM-TIME TIME INTEGER 0 THE-TANK
            ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
          with-data: NIL connection-state: NOT-CONSUMED
        connection CON-1
          links:
            CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
              export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
```

```
            APP-EXEC
          to-imp:
          SIM-TIME TIME INTEGER 0 THE-THROTTLE
            ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
          with-data: NIL connection-state: NOT-CONSUMED
      connection CON-2
        links:
          CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
            export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
              APP-EXEC
          to-imp:
          SIM-TIME TIME INTEGER 0 THE-ENGINE
            ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
          with-data: NIL connection-state: NOT-CONSUMED
      connection CON-3
        links:
          OUT1 SIGNAL BOOLEAN NIL ENGINE-START
            export-path: ( SUB-1 ) export-owner: SUB-1
          to-imp:
          START SIGNAL BOOLEAN NIL THE-ENGINE
            ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
          with-data: NIL connection-state: NOT-CONSUMED
      connection CON-4
        links:
          OUT1 SIGNAL BOOLEAN NIL FUEL-PUMP-START
            export-path: ( SUB-1 ) export-owner: SUB-1
          to-imp:
          START SIGNAL BOOLEAN NIL THE-TANK
            ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
          with-data: NIL connection-state: NOT-CONSUMED
  THE-TANK THE-THROTTLE THE-ENGINE ENGINE-START
  FUEL-PUMP-START SUB-1 SUB-2 GLOBAL-TIMER EVENT-HANDLER
  CONN-MGR APP-EXEC TEST-ME
.> (ar 16)
(* Once again, the faultly semantic chaecks fail *)
ERROR -- "No subsystem produces data of category PERCENT for object THE-THROTTLE"
Object: subsystem SUB-2 is controls:
  THE-TANK, THE-THROTTLE, THE-ENGINE
  imports:
    START SIGNAL BOOLEAN NIL THE-TANK
      ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
      START SIGNAL BOOLEAN NIL THE-ENGINE
        ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
      THROTTLE-INDEX PERCENT REAL 0.25 THE-THROTTLE
        ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
      CONSUMPTION-RATE FLOW-RATE REAL 0.0 THE-TANK
        ( FUEL-FLOW-RATE SUB-2 THE-ENGINE
        ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
      FUEL-AVAILABLE? SIGNAL BOOLEAN NIL THE-THROTTLE
        ( FUEL-AVAILABLE? SUB-2 THE-TANK
        ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
      INFLOW-RATE FLOW-RATE REAL 0.0 THE-ENGINE
        ( REQUESTED-FLOW-RATE SUB-2 THE-THROTTLE
        ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
      SIM-TIME TIME INTEGER 0 THE-TANK
        ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
```

```
      SIM-TIME TIME INTEGER 0 THE-THROTTLE
        ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
      SIM-TIME TIME INTEGER 0 THE-ENGINE
        ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
  exports:
    FUEL-AVAILABLE? SIGNAL BOOLEAN NIL THE-TANK
      export-path: ( SUB-2 ) export-owner: SUB-2
      FUEL-TANK-WEIGHT FORCE REAL 150.0 THE-TANK
        export-path: ( SUB-2 ) export-owner: SUB-2
      REQUESTED-FLOW-RATE FLOW-RATE REAL 0.0 THE-THROTTLE
        export-path: ( SUB-2 ) export-owner: SUB-2
      THRUST FORCE REAL 0.0 THE-ENGINE
        export-path: ( SUB-2 ) export-owner: SUB-2
      FUEL-FLOW-RATE FLOW-RATE REAL 0.0 THE-ENGINE
        export-path: ( SUB-2 ) export-owner: SUB-2
  initialize procedure: update procedure:
  update THE-TANK update THE-THROTTLE update THE-ENGINE

Rule successfully applied.
application definition TD-TEST
  execution-mode: TIME-DRIVEN-SEQUENTIAL
    connected-by
      connection CON-0
        links:
          CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
            export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
              APP-EXEC
          to-imp:
          SIM-TIME TIME INTEGER 0 THE-TANK
            ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
          with-data: NIL connection-state: NOT-CONSUMED
        connection CON-1
          links:
            CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
              export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
                APP-EXEC
            to-imp:
            SIM-TIME TIME INTEGER 0 THE-THROTTLE
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
            with-data: NIL connection-state: NOT-CONSUMED
        connection CON-2
          links:
            CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
              export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
                APP-EXEC
            to-imp:
            SIM-TIME TIME INTEGER 0 THE-ENGINE
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
            with-data: NIL connection-state: NOT-CONSUMED
        connection CON-3
          links:
            OUT1 SIGNAL BOOLEAN NIL ENGINE-START
              export-path: ( SUB-1 ) export-owner: SUB-1
            to-imp:
            START SIGNAL BOOLEAN NIL THE-ENGINE
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
```

```
          with-data: NIL connection-state: NOT-CONSUMED
        connection CON-4
          links:
            OUT1 SIGNAL BOOLEAN NIL FUEL-PUMP-START
              export-path: ( SUB-1 ) export-owner: SUB-1
            to-imp:
            START SIGNAL BOOLEAN NIL THE-TANK
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
            with-data: NIL connection-state: NOT-CONSUMED
    THE-TANK THE-THROTTLE THE-ENGINE ENGINE-START
    FUEL-PUMP-START SUB-1 SUB-2 GLOBAL-TIMER EVENT-HANDLER
    CONN-MGR APP-EXEC TEST-ME
.> (ar 15)
Rule failed to apply.
.> (ar 17)
Rule successfully applied.
application definition TD-TEST
  execution-mode: TIME-DRIVEN-SEQUENTIAL
    connected-by
      connection CON-0
        links:
          CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
            export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
              APP-EXEC
          to-imp:
          SIM-TIME TIME INTEGER 0 THE-TANK
            ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
          with-data: NIL connection-state: NOT-CONSUMED
        connection CON-1
          links:
            CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
              export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
                APP-EXEC
            to-imp:
            SIM-TIME TIME INTEGER 0 THE-THROTTLE
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
            with-data: NIL connection-state: NOT-CONSUMED
        connection CON-2
          links:
            CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
              export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
                APP-EXEC
            to-imp:
            SIM-TIME TIME INTEGER 0 THE-ENGINE
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
            with-data: NIL connection-state: NOT-CONSUMED
        connection CON-3
          links:
            OUT1 SIGNAL BOOLEAN NIL ENGINE-START
              export-path: ( SUB-1 ) export-owner: SUB-1
            to-imp:
            START SIGNAL BOOLEAN NIL THE-ENGINE
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
            with-data: NIL connection-state: NOT-CONSUMED
        connection CON-4
          links:
```

```
            OUT1 SIGNAL BOOLEAN NIL FUEL-PUMP-START
              export-path: ( SUB-1 ) export-owner: SUB-1
            to-imp:
            START SIGNAL BOOLEAN NIL THE-TANK
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
            with-data: NIL connection-state: NOT-CONSUMED
    THE-TANK THE-THROTTLE THE-ENGINE ENGINE-START
    FUEL-PUMP-START SUB-1 SUB-2 GLOBAL-TIMER EVENT-HANDLER
    CONN-MGR APP-EXEC TEST-ME
.> (ar 15)
The current simulation time is  0
The current simulation time is  1
The current simulation time is  2
The current simulation time is  3
The current simulation time is  4
scavenging...done
A fuel-flow sequencing error has occurred  (* At time 4, when the engine and   *)
The current simulation time is  5          (* fuel tank update, they see that  *)
The current simulation time is  6          (* the switches have been turned on *)
The current simulation time is  7          (* at the same time                 *)
The current simulation time is  8
The current simulation time is  9

 Execution Stopped
Rule successfully applied.

(* This is the state following execution *)
application definition TD-TEST
  execution-mode: TIME-DRIVEN-SEQUENTIAL
    connected-by
      connection CON-0
        links:
          CURRENT-TIME TIME INTEGER 9 GLOBAL-TIMER
            export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
              APP-EXEC
          to-imp:
          SIM-TIME TIME INTEGER 8 THE-TANK
            ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
          with-data: 8 connection-state: CONSUMED
        connection CON-1
          links:
            CURRENT-TIME TIME INTEGER 9 GLOBAL-TIMER
              export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
                APP-EXEC
            to-imp:
            SIM-TIME TIME INTEGER 8 THE-THROTTLE
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
            with-data: 8 connection-state: CONSUMED
        connection CON-2
          links:
            CURRENT-TIME TIME INTEGER 9 GLOBAL-TIMER
              export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
                APP-EXEC
            to-imp:
            SIM-TIME TIME INTEGER 8 THE-ENGINE
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
```

```
          with-data: 8 connection-state: CONSUMED
        connection CON-3
          links:
            OUT1 SIGNAL BOOLEAN T ENGINE-START
              export-path: ( SUB-1 ) export-owner: SUB-1
            to-imp:
            START SIGNAL BOOLEAN T THE-ENGINE
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
            with-data: T connection-state: CONSUMED
        connection CON-4
          links:
            OUT1 SIGNAL BOOLEAN T FUEL-PUMP-START
              export-path: ( SUB-1 ) export-owner: SUB-1
            to-imp:
            START SIGNAL BOOLEAN T THE-TANK
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
            with-data: T connection-state: CONSUMED
    THE-TANK THE-THROTTLE THE-ENGINE ENGINE-START
    FUEL-PUMP-START SUB-1 SUB-2 GLOBAL-TIMER EVENT-HANDLER
    CONN-MGR APP-EXEC TEST-ME

(* This is the Old-Events List *)

.> (ar 18)
start-event EVENT1 ev-time: 0 for-primitive: EVENT-HANDLER
  through-subsystems: APP-EXEC priority: 100 start-time: 0

transmit-event CLOCK-UPDATE ev-time: 0 from-primitive:
  GLOBAL-TIMER in-subsystem: APP-EXEC from-export:
  CURRENT-TIME priority: 50

receive-event RX-2 ev-time: 0 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 0 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 0 for-primitive: THE-TANK
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

transmit-event CLOCK-UPDATE ev-time: 1 from-primitive:
  GLOBAL-TIMER in-subsystem: APP-EXEC from-export:
  CURRENT-TIME priority: 50

receive-event RX-2 ev-time: 1 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 1 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 1 for-primitive: THE-TANK
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

scavenging...done
update-event EVENT2 ev-time: 1 for-primitive: THE-TANK
  through-subsystems: SUB-2 priority: 1
```

```
transmit-event *UNDEFINED* ev-time: 1 from-primitive:
  THE-TANK in-subsystem: SUB-2 from-export: FUEL-TANK-WEIGHT
  priority: 50

update-event EVENT3 ev-time: 1 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 1 from-primitive:
  THE-THROTTLE in-subsystem: SUB-2 from-export:
  REQUESTED-FLOW-RATE priority: 50

update-event EVENT4 ev-time: 1 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 1 from-primitive:
  THE-ENGINE in-subsystem: SUB-2 from-export: THRUST
  priority: 50

transmit-event CLOCK-UPDATE ev-time: 2 from-primitive:
  GLOBAL-TIMER in-subsystem: APP-EXEC from-export:
  CURRENT-TIME priority: 50

receive-event RX-2 ev-time: 2 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 2 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 2 for-primitive: THE-TANK
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

update-event EVENT2 ev-time: 2 for-primitive: THE-TANK
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 2 from-primitive:
  THE-TANK in-subsystem: SUB-2 from-export: FUEL-TANK-WEIGHT
  priority: 50

update-event EVENT3 ev-time: 2 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 2 from-primitive:
  THE-THROTTLE in-subsystem: SUB-2 from-export:
  REQUESTED-FLOW-RATE priority: 50

update-event EVENT4 ev-time: 2 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 2 from-primitive:
  THE-ENGINE in-subsystem: SUB-2 from-export: THRUST
  priority: 50

transmit-event CLOCK-UPDATE ev-time: 3 from-primitive:
  GLOBAL-TIMER in-subsystem: APP-EXEC from-export:
  CURRENT-TIME priority: 50
```

```
receive-event RX-2 ev-time: 3 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 3 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 3 for-primitive: THE-TANK
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

set-state-event EVENT5 ev-time: 3 for-primitive:
  ENGINE-START attr-values: (POSITION, ON)
  through-subsystems: SUB-1 priority: 1

transmit-event *UNDEFINED* ev-time: 3 from-primitive:
  ENGINE-START in-subsystem: SUB-1 from-export: OUT1
  priority: 50

receive-event RX-2 ev-time: 3 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 to-import START priority: 10

set-state-event EVENT6 ev-time: 3 for-primitive:
  FUEL-PUMP-START attr-values: (POSITION, ON)
  through-subsystems: SUB-1 priority: 1

transmit-event *UNDEFINED* ev-time: 3 from-primitive:
  FUEL-PUMP-START in-subsystem: SUB-1 from-export: OUT1
  priority: 50

receive-event RX-2 ev-time: 3 for-primitive: THE-TANK
  through-subsystems: SUB-2 to-import START priority: 10

transmit-event CLOCK-UPDATE ev-time: 4 from-primitive:
  GLOBAL-TIMER in-subsystem: APP-EXEC from-export:
  CURRENT-TIME priority: 50

receive-event RX-2 ev-time: 4 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 4 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 4 for-primitive: THE-TANK
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

update-event EVENT2 ev-time: 4 for-primitive: THE-TANK
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 4 from-primitive:
  THE-TANK in-subsystem: SUB-2 from-export: FUEL-TANK-WEIGHT
  priority: 50

update-event EVENT3 ev-time: 4 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 4 from-primitive:
```

```
  THE-THROTTLE in-subsystem: SUB-2 from-export:
  REQUESTED-FLOW-RATE priority: 50

update-event EVENT4 ev-time: 4 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 4 from-primitive:
  THE-ENGINE in-subsystem: SUB-2 from-export: THRUST
  priority: 50

transmit-event CLOCK-UPDATE ev-time: 5 from-primitive:
  GLOBAL-TIMER in-subsystem: APP-EXEC from-export:
  CURRENT-TIME priority: 50

receive-event RX-2 ev-time: 5 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 5 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 5 for-primitive: THE-TANK
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

update-event EVENT2 ev-time: 5 for-primitive: THE-TANK
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 5 from-primitive:
  THE-TANK in-subsystem: SUB-2 from-export: FUEL-TANK-WEIGHT
  priority: 50

update-event EVENT3 ev-time: 5 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 5 from-primitive:
  THE-THROTTLE in-subsystem: SUB-2 from-export:
  REQUESTED-FLOW-RATE priority: 50

update-event EVENT4 ev-time: 5 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 5 from-primitive:
  THE-ENGINE in-subsystem: SUB-2 from-export: THRUST
  priority: 50

transmit-event CLOCK-UPDATE ev-time: 6 from-primitive:
  GLOBAL-TIMER in-subsystem: APP-EXEC from-export:
  CURRENT-TIME priority: 50

receive-event RX-2 ev-time: 6 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 6 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 6 for-primitive: THE-TANK
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10
```

```
update-event EVENT2 ev-time: 6 for-primitive: THE-TANK
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 6 from-primitive:
  THE-TANK in-subsystem: SUB-2 from-export: FUEL-TANK-WEIGHT
  priority: 50

update-event EVENT3 ev-time: 6 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 6 from-primitive:
  THE-THROTTLE in-subsystem: SUB-2 from-export:
  REQUESTED-FLOW-RATE priority: 50

update-event EVENT4 ev-time: 6 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 6 from-primitive:
  THE-ENGINE in-subsystem: SUB-2 from-export: THRUST
  priority: 50

transmit-event CLOCK-UPDATE ev-time: 7 from-primitive:
  GLOBAL-TIMER in-subsystem: APP-EXEC from-export:
  CURRENT-TIME priority: 50

receive-event RX-2 ev-time: 7 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 7 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 7 for-primitive: THE-TANK
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

update-event EVENT2 ev-time: 7 for-primitive: THE-TANK
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 7 from-primitive:
  THE-TANK in-subsystem: SUB-2 from-export: FUEL-TANK-WEIGHT
  priority: 50

update-event EVENT3 ev-time: 7 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 7 from-primitive:
  THE-THROTTLE in-subsystem: SUB-2 from-export:
  REQUESTED-FLOW-RATE priority: 50

update-event EVENT4 ev-time: 7 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 7 from-primitive:
  THE-ENGINE in-subsystem: SUB-2 from-export: THRUST
  priority: 50
```

```
transmit-event CLOCK-UPDATE ev-time: 8 from-primitive:
  GLOBAL-TIMER in-subsystem: APP-EXEC from-export:
  CURRENT-TIME priority: 50

receive-event RX-2 ev-time: 8 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 8 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

receive-event RX-2 ev-time: 8 for-primitive: THE-TANK
  through-subsystems: SUB-2 to-import SIM-TIME priority: 10

update-event EVENT2 ev-time: 8 for-primitive: THE-TANK
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 8 from-primitive:
  THE-TANK in-subsystem: SUB-2 from-export: FUEL-TANK-WEIGHT
  priority: 50

update-event EVENT3 ev-time: 8 for-primitive: THE-THROTTLE
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 8 from-primitive:
  THE-THROTTLE in-subsystem: SUB-2 from-export:
  REQUESTED-FLOW-RATE priority: 50

update-event EVENT4 ev-time: 8 for-primitive: THE-ENGINE
  through-subsystems: SUB-2 priority: 1

transmit-event *UNDEFINED* ev-time: 8 from-primitive:
  THE-ENGINE in-subsystem: SUB-2 from-export: THRUST
  priority: 50

stop-event EVENT3 ev-time: 9 for-primitive: EVENT-HANDLER
  through-subsystems: APP-EXEC priority: 100 stop-time: 9

Rule successfully applied.
application definition TD-TEST
  execution-mode: TIME-DRIVEN-SEQUENTIAL
    connected-by
      connection CON-0
        links:
          CURRENT-TIME TIME INTEGER 9 GLOBAL-TIMER
            export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
              APP-EXEC
          to-imp:
          SIM-TIME TIME INTEGER 8 THE-TANK
            ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
          with-data: 8 connection-state: CONSUMED
        connection CON-1
          links:
            CURRENT-TIME TIME INTEGER 9 GLOBAL-TIMER
              export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
                APP-EXEC
            to-imp:
```

```
            SIM-TIME TIME INTEGER 8 THE-THROTTLE
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
            with-data: 8 connection-state: CONSUMED
        connection CON-2
          links:
            CURRENT-TIME TIME INTEGER 9 GLOBAL-TIMER
              export-path: ( APP-EXEC GLOBAL-TIMER ) export-owner:
                APP-EXEC
            to-imp:
            SIM-TIME TIME INTEGER 8 THE-ENGINE
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
            with-data: 8 connection-state: CONSUMED
        connection CON-3
          links:
            OUT1 SIGNAL BOOLEAN T ENGINE-START
              export-path: ( SUB-1 ) export-owner: SUB-1
            to-imp:
            START SIGNAL BOOLEAN T THE-ENGINE
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
            with-data: T connection-state: CONSUMED
        connection CON-4
          links:
            OUT1 SIGNAL BOOLEAN T FUEL-PUMP-START
              export-path: ( SUB-1 ) export-owner: SUB-1
            to-imp:
            START SIGNAL BOOLEAN T THE-TANK
              ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
            with-data: T connection-state: CONSUMED
    THE-TANK THE-THROTTLE THE-ENGINE ENGINE-START
    FUEL-PUMP-START SUB-1 SUB-2 GLOBAL-TIMER EVENT-HANDLER
    CONN-MGR APP-EXEC TEST-ME
.>
.> (pn 'sub-1)
subsystem SUB-1 is controls: ENGINE-START, FUEL-PUMP-START
  imports:
  exports:
    OUT1 SIGNAL BOOLEAN T ENGINE-START
      export-path: ( SUB-1 ) export-owner: SUB-1
      OUT1 SIGNAL BOOLEAN T FUEL-PUMP-START
        export-path: ( SUB-1 ) export-owner: SUB-1
  initialize procedure: update procedure:
  update ENGINE-START update FUEL-PUMP-START
.> (pn 'sub-2)
subsystem SUB-2 is controls:
  THE-TANK, THE-THROTTLE, THE-ENGINE
  imports:
    START SIGNAL BOOLEAN T THE-TANK
      ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
      START SIGNAL BOOLEAN T THE-ENGINE
        ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
      THROTTLE-INDEX PERCENT REAL 0.25 THE-THROTTLE
        ( ) import-path: ( SUB-2 ) import-owner: SUB-2 not-changed
      CONSUMPTION-RATE FLOW-RATE REAL 0.05 THE-TANK
        ( FUEL-FLOW-RATE SUB-2 THE-ENGINE
        ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
      FUEL-AVAILABLE? SIGNAL BOOLEAN T THE-THROTTLE
```

```
      ( FUEL-AVAILABLE? SUB-2 THE-TANK
      ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
    INFLOW-RATE FLOW-RATE REAL 0.05 THE-ENGINE
      ( REQUESTED-FLOW-RATE SUB-2 THE-THROTTLE
      ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
    SIM-TIME TIME INTEGER 8 THE-TANK
      ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
    SIM-TIME TIME INTEGER 8 THE-THROTTLE
      ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
    SIM-TIME TIME INTEGER 8 THE-ENGINE
      ( ) import-path: ( SUB-2 ) import-owner: SUB-2 is-changed
  exports:
    FUEL-AVAILABLE? SIGNAL BOOLEAN T THE-TANK
      export-path: ( SUB-2 ) export-owner: SUB-2
      FUEL-TANK-WEIGHT FORCE REAL 149.84999 THE-TANK
        export-path: ( SUB-2 ) export-owner: SUB-2
      REQUESTED-FLOW-RATE FLOW-RATE REAL 0.05 THE-THROTTLE
        export-path: ( SUB-2 ) export-owner: SUB-2
      THRUST FORCE REAL 50.0 THE-ENGINE
        export-path: ( SUB-2 ) export-owner: SUB-2
      FUEL-FLOW-RATE FLOW-RATE REAL 0.05 THE-ENGINE
        export-path: ( SUB-2 ) export-owner: SUB-2
  initialize procedure: update procedure:
  update THE-TANK update THE-THROTTLE update THE-ENGINE
.>
```

B.4 Conclusion

The tests all caused the predicted behavior to occur. They do not represent an all-encompassing set of possible applications, but they are sufficient to demonstrate that the executive correctly services all events. The sample tests and results contained herein show how the kinds of results obtained from the testing of the application executive. More details on application testing are contained in Waggoner's thesis (24).

Appendix C. Application Executive Instances

C.1 Introduction

Architect's application executive is a subsystem which controls primitives which were discovered during domain analysis. This appendix presents two executive subsystems expressed in the OCU architecture specific language (ASL).

C.2 Event-Driven Sequential Executive Subsystem

```
%
% filename: ed-seq-exec
% author  : Bob Welgan
% date    : 30 Sep 93
%
% This file, written in OCU language, is designed to be parsed into
% the Refine object base.  It defines the event-driven sequential application
% executive subsystem.

  event-driven-clock  global-timer
    is-at-time: 0

  event-driven-sequential-event-manager  event-handler
    manages:  start-event event1
              ev-time: 0
              for-primitive: event-handler
              through-subsystems: app-exec
              priority: 100
              start-time: 1,

              stop-event event2
              ev-time: 50
              for-primitive: event-handler
              through-subsystems: app-exec
              priority: 100
              stop-time: 50

  connection-manager conn-mgr
    manages: connection con-0

  subsystem app-exec is

    controls: global-timer, event-handler, conn-mgr

    imports: SIMULATION-TIME TIME INTEGER 0 EVENT-HANDLER
             (CURRENT-TIME APP-EXEC GLOBAL-TIMER)
             import-path: (APP-EXEC)
             import-owner:  APP-EXEC is-changed

             NEW-EVENT AN-EVENT SET-EVENT-OBJ 0 EVENT-HANDLER
             (NEW-EVENT APP-EXEC CONN-MGR)
             import-path:  (APP-EXEC)
```

```
            import-owner:  APP-EXEC is-changed

            NEW-TIME TIME INTEGER 0 GLOBAL-TIMER
            (SIMULATION-TIME APP-EXEC EVENT-HANDLER)
            import-path:  (APP-EXEC)
            import-owner:  APP-EXEC is-changed

            CURR-EVENT AN-EVENT EVENT-OBJ 0 CONN-MGR
            (CURR-EVENT APP-EXEC EVENT-HANDLER)
            import-path: (APP-EXEC)
            import-owner: APP-EXEC is-changed

    exports: SIMULATION-TIME TIME INTEGER 0 EVENT-HANDLER
            export-path: (APP-EXEC)
            export-owner: APP-EXEC

            CURR-EVENT AN-EVENT EVENT-OBJ 0 EVENT-HANDLER
            export-path: (APP-EXEC)
            export-owner: APP-EXEC

            CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
            export-path: (APP-EXEC)
            export-owner: APP-EXEC

            NEW-EVENT AN-EVENT SET-EVENT-OBJ 0 CONN-MGR
            export-path: (APP-EXEC)
            export-owner: APP-EXEC

    update procedure:
      update global-timer
      update event-handler
      update conn-mgr
```

C.3 Time-Driven Sequential Executive Subsystem

```
%
% filename: td-solo
% author  : Bob Welgan
% date    : 18 Oct 93
%
% This file, written in OCU language, is designed to be parsed into
% the Refine object base.  It defines the time-driven sequential application
% executive subsystem.

  time-driven-clock  global-timer
    is-at-time: 0

  time-driven-sequential-event-manager  event-handler
    manages:  start-event event1
              ev-time: 0
              for-primitive: event-handler
              through-subsystems: app-exec
              priority: 100
              start-time: 1,
```

```
            stop-event event2
            ev-time: 20
            for-primitive: event-handler
            through-subsystems: app-exec
            priority: 100
            stop-time: 20

connection-manager conn-mgr
  manages: connection con-0

subsystem app-exec is

  controls: global-timer, event-handler, conn-mgr

  imports: SIMULATION-TIME TIME INTEGER 0 EVENT-HANDLER
           (CURRENT-TIME APP-EXEC GLOBAL-TIMER)
           import-path: (APP-EXEC)
           import-owner:  APP-EXEC is-changed

           NEW-EVENT AN-EVENT SET-EVENT-OBJ 0 EVENT-HANDLER
           (NEW-EVENT APP-EXEC CONN-MGR,
            NEW-EVENT APP-EXEC GLOBAL-TIMER)
           import-path:  (APP-EXEC)
           import-owner:  APP-EXEC is-changed

           NEW-TIME TIME INTEGER 0 GLOBAL-TIMER
           (SIMULATION-TIME APP-EXEC EVENT-HANDLER)
           import-path:  (APP-EXEC)
           import-owner:  APP-EXEC is-changed

           CURR-EVENT AN-EVENT EVENT-OBJ 0 CONN-MGR
           (CURR-EVENT APP-EXEC EVENT-HANDLER)
           import-path: (APP-EXEC)
           import-owner: APP-EXEC is-changed

  exports: SIMULATION-TIME TIME INTEGER 0 EVENT-HANDLER
           export-path: (APP-EXEC)
           export-owner: APP-EXEC

           CURR-EVENT AN-EVENT EVENT-OBJ 0 EVENT-HANDLER
           export-path: (APP-EXEC)
           export-owner: APP-EXEC

           CURRENT-TIME TIME INTEGER 0 GLOBAL-TIMER
           export-path: (APP-EXEC)
           export-owner: APP-EXEC

           NEW-EVENT AN-EVENT SET-EVENT-OBJ 0 CONN-MGR
           export-path: (APP-EXEC)
           export-owner: APP-EXEC

           NEW-EVENT AN-EVENT SET-EVENT-OBJ 0 GLOBAL-TIMER
           export-path: (APP-EXEC GLOBAL-TIMER)
           export-owner: APP-EXEC
```

```
update procedure:
  update global-timer
  update event-handler
  update conn-mgr
```

C.4 Conclusion

These subsystems enable Architect to execute applications in time-driven and event-driven sequential modes.

Bibliography

1. Abraham Silberschatz, James L. Peterson, Peter Galvin. *Operating System Concepts, 3rd edition*. New York, NY: Addison-Wesley Publishing Company, Inc., 1991.
2. Anderson, Cindy Griffin. *Creating and Manipulating Formalized Software Architectures to Support a Domain-Oriented Application and Composition System*. MS thesis, AFIT/GCS/ENG/92D-01, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992 (AD-A258900).
3. Bailor, Maj. Paul D. "A Framework for Application Executives for OCU-Based Software Architectures." Proposed structure of OCU Model application executive, September 1992.
4. Ben Potter, Jane Sinclair and David Till. *An Introduction to Formal Specification and Z*. Prentice Hall International (UK) Ltd., 1991.
5. Cossentine, Jay. *Developing a Sophisticated User Interface to Support the Architect Domain-Oriented Application Composition and Generation Systems*. MS thesis, AFIT/GCS/ENG/93D-04, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.
6. D'Ippolito, Richard S. "Using Models in Software Engineering." *Proceedings: TRI-Ada '89*. 256–265. New York, NY: Association of Computing Machinery, Inc., 1989.
7. Fugimoto, Richard M. "Parallel Discrete Event Simulation," *Communications of the ACM*, *33*(10):31–53 (October 1990).
8. Gool, Warren Evan. *Alternative Architectures for Domain-Oriented Application Composition and Generation Systems*. MS thesis, AFIT/GCS/ENG/93D-11, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.
9. Gregory R. Andrews, Fred B. Scheider. "Concepts and Notations for Concurrent Programming," *Computing Surveys*, *15*(1):3–43 (March 1983).
10. Hall, Anthony. "Seven Myths of Formal Methods," *IEEE Software*, 11–19 (September 1990).
11. James Rumbaugh, et al. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991.
12. Lee, Kenneth J. and others. *Model-Based Software Development (Draft)*. Technical Report CMU/SEI-92-SR-00, Software Engineering Institute, December 1991.
13. Levi, Shem-Tov and Ashok K. Agrawala. *Real-Time System Design*. McGraw-Hill, 1990.
14. Lowry, M.R. *Software Engineering in the Twenty-first Century*, chapter 24, 600–680. MIT Press, 1991.
15. Paul D. Bailor, others. "Summary of J-MASS Research." Formal Methods Research Group Proposal, December 1992.
16. Prieto-Diaz, Ruben. "Domain Analysis for Reusability." *Proceedings of the COMPSAC '87*. 23–29. 1987.

17. Prieto-Diaz, Ruben. "Domain Analysis: An Introduction," *Software Engineering Notes*, *15*(2):47–45 (April 1990).

18. Randour, Mary Ann. *Creating and Manipulating a Domain-Specific Formal Object Base to Support a Domain-Oriented Application and Composition System*. MS thesis, AFIT/GCS/ENG/92D-13, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992 (AD-A259018).

19. Reasoning Systems, Inc. *DIALECT User's Guide*. Palo Alto, CA, July 1990.

20. Reasoning Systems, Inc. *REFINE User's Guide*. Palo Alto, CA, May 1990.

21. Science Applications International Corporation, Wright Laboratory Avionics Directorate. *Simulation Support Environment Architectural Design Team Report, Version 2.0*, February 1993.

22. "Bugs in the Program: Problems in Federal Government Computer Software Development and Regulation." Committee on Science, Space, and Technology, U.S. House of Representatives, December 1989.

23. V.S. Alagar, G. Ramanathan. "Functional Specification and Proof of Correctness for Time Dependant Behaviour of Reactive Systems," *Formal Aspects of Computing*, *3*(3):253–283 (Jul-Sep 1991).

24. Waggoner, Robert W. *Domain Modeling of Time-Dependent Systems*. MS thesis, AFIT/GCS/ENG/93D-23, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

25. Warner, Russell M. *A Method for Populating the Knowledge Base of AFIT's Domain-Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/93D-24, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

26. Warner, Russell M. "The Theory Behind Software Application Generation at AFIT." COMM 680 Briefing, May 93.

27. Whitted, Gary A. *Software Development Plan for the J-MASS Modeling Components - Vol II*. Architectural Technical Working Group, ASD/RWWW, April 1992.

28. Will Tracz, Lou Coglianese and Patrick Young. "A Domain-Specific Software Architecture Engineering Process." From Collected Papers of the Domain-Specific Software Architectures (DSSA) Avionics Domain Application Environment (ADAGE), May 93.

29. Young, Frank Charles Duane. *Creating a Real Time System Using Reacto*. MS thesis, AFIT/ENG/GCS/92D-24, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992 (AD-A259224).

Vita

Captain Bob Welgan was born on May 21, 1965 in Dover, Delaware. He attended Caesar Rodney Senior High School in nearby Camden, Delaware. Captain Welgan attended the University of Maryland, College Park after graduation from high school in 1983. In 1984, he entered the United States Air Force Academy, where he earned a Bachelor of Science in Computer Science. From 1988 until 1992, Captain Welgan worked in the Space Surveillance Program Office at Los Angeles AFB, California. He entered the Air Force Institute of Technology in May 1992.

Permanent address: 222 North State Street
Dover, Delaware 19901

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December, 1993	Master's Thesis

4. TITLE AND SUBTITLE
DOMAIN ANALYSIS AND MODELING
OF A MODEL-BASED SOFTWARE EXECUTIVE

5. FUNDING NUMBERS

6. AUTHOR(S)
Robert L. Welgan, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/93D-25

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Capt Rick Painter
2241 Avionics Circle, Suite 16
WL/AAWA-1 BLD 620
Wright-Patterson AFB, OH 45433-7765

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research adapted the domain analysis techniques of Prieto-Díaz and Tracz to specify a domain analysis process which was used to conduct domain analysis over the domain of software executives. This analysis created a set of informal and formal domain model artifacts. The domain model artifacts were instantiated into two application executive subsystems. These executive subsystems operated in Architect, a domain-oriented application composition system based on the Object-Connection-Update (OCU) model. This research demonstrated and evaluated execution of the instantiated executive domain model in a series of event-driven and time-driven applications. As a consequence of developing the application executive for Architect, this research proposes additions to the OCU model.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software Engineering, Operating Systems, Knowledge Based Systems, Domain Modeling, Domain-Specific Languages, Application Composition Systems	158
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.